

Advanced Data Structures and Algorithm Analysis

丁尧相
浙江大学

Fall & Winter 2025
Lecture I

Outline: Balanced Search Trees (I)

- Binary search trees
- AVL trees
- Splay trees
- Amortized analysis
- Take-home messages

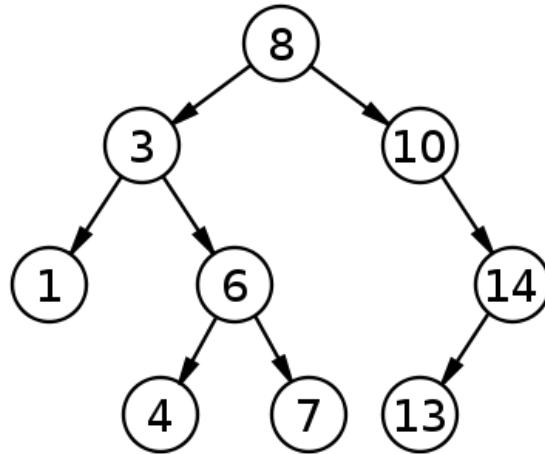
Outline: Balanced Search Trees (I)

- Binary search trees
- AVL trees
- Splay trees
- Amortized analysis
- Take-home messages

Data Structures

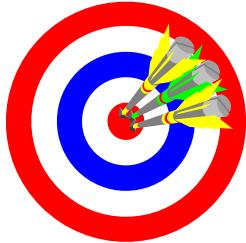
- Data structures represent **dynamic sets** of instances.
 - dynamic means the set can change.
 - can be ordered or unordered.
- Data structures are **abstractions**: supporting group of operations:
 - queries:
 - search, minimum, maximum, successor, predecessor...
 - modifying operations:
 - insert, delete...
- A proper data structure effectively speeds up the set operations.
in terms of the size of the data structure

Binary Search Trees (BSTs)



- Every node has at most two children.
- The key value for the left child is smaller, and the right child is larger.
- The tree operations (search, insert, delete, minimum, maximum, successor, predecessor...) have time costs closely related to **tree depth**.
- Balancing is to reduce tree depth in order to reduce time costs.

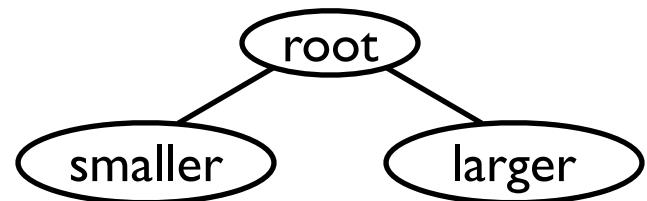
Balanced BSTs



Target : Speed up searching (with insertion and deletion)

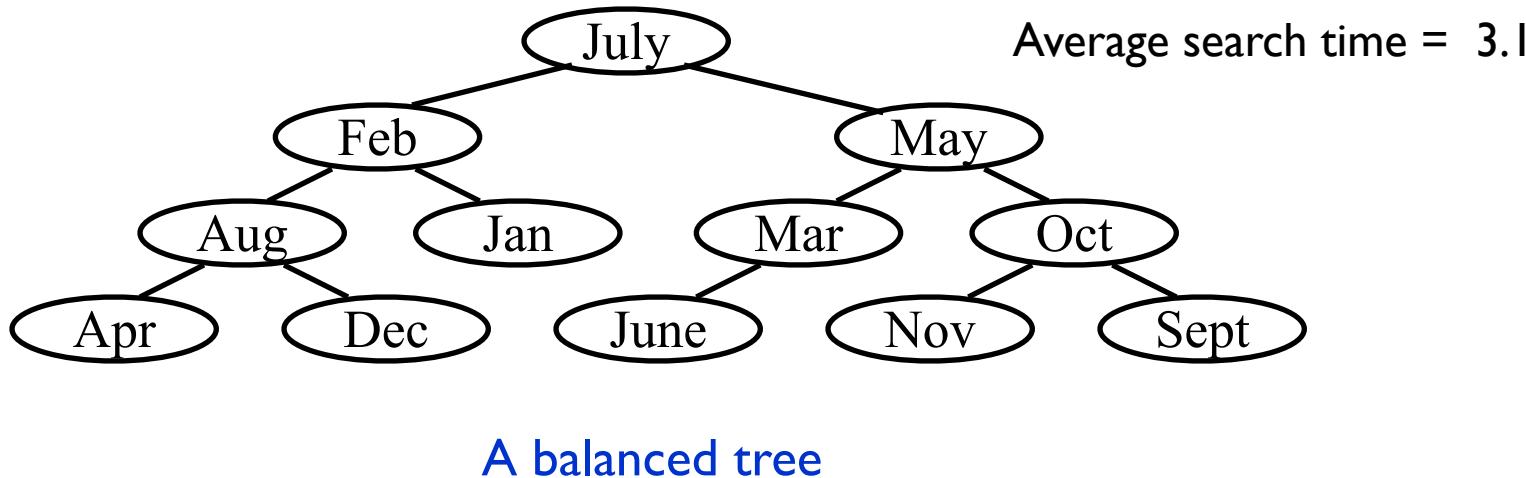
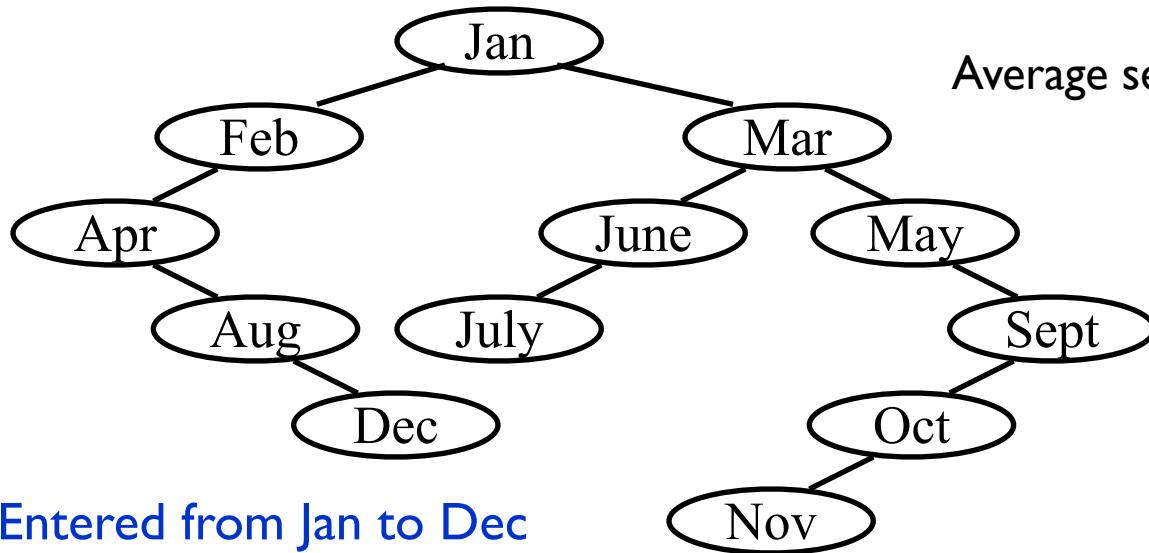


Tool : Binary search trees

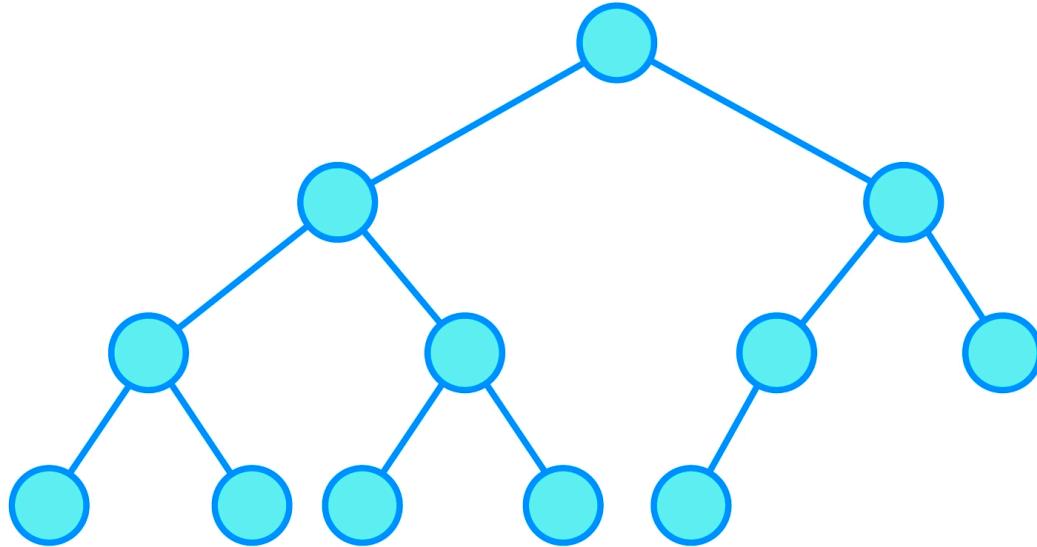


Problem : Although $T_p = O(\text{height})$, but the height can be as bad as $O(N)$.

【Example】 2 binary search trees obtained for the months of the year



Why Not Use Complete BST?

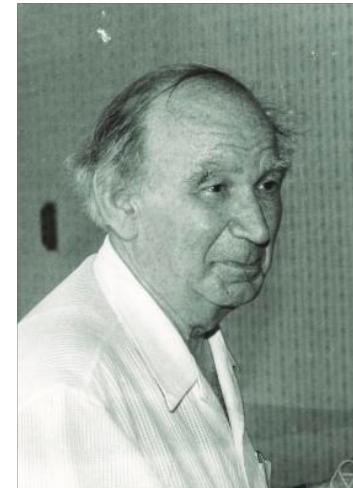


The constraint is too strong.
If the complete constraint is to be preserved,
every operation involves global tuning of the structure.
We should relax the constraint.

Outline: Balanced Binary Search Trees (I)

- Binary search trees
- AVL trees
- Splay trees
- Amortized analysis
- Take-home messages

Adelson-Velskii-Landis (AVL) Trees (1962)



- Self-balanced trees **dynamically modify tree structure** to keep the tree balanced during operations.

Figure courtesy: https://www.chessprogramming.org/Georgy_Adelson-Velsky
https://en.wikipedia.org/wiki/Evgenii_Landis

Adelson-Velskii-Landis (AVL) Trees (1962)



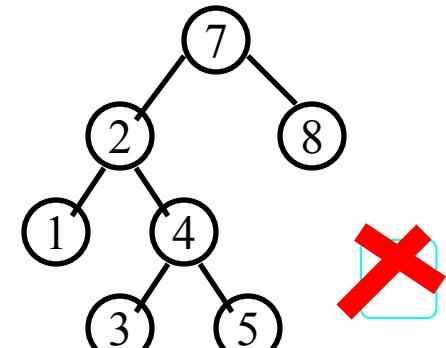
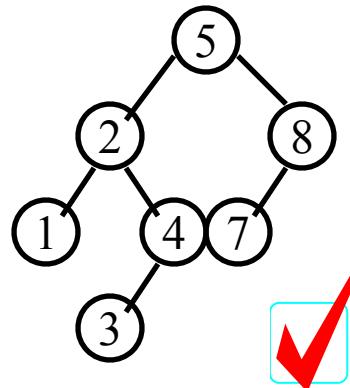
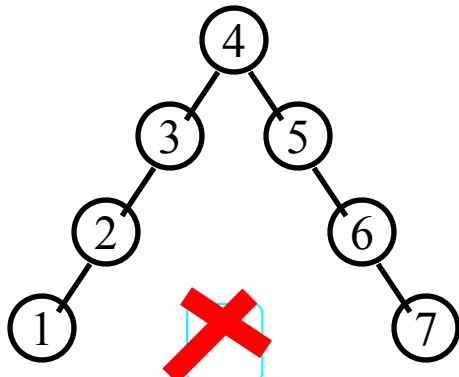
AVL Trees

The height of an empty tree is defined to be -1 .

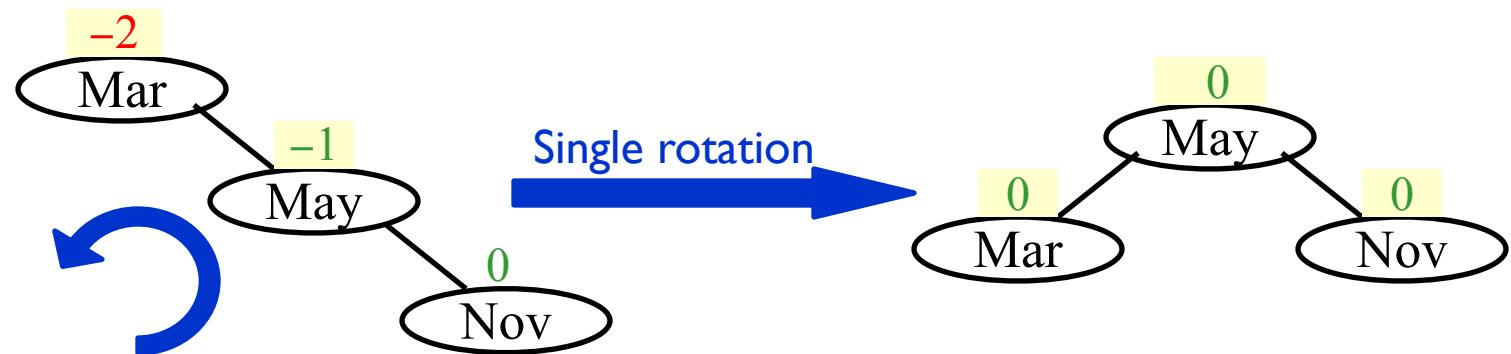
【Definition】 An empty binary tree is height-balanced. If T is a nonempty binary tree with T_L and T_R as its left and right subtrees, then T is **height-balanced** iff

- (1) T_L and T_R are height-balanced, and
- (2) $|h_L - h_R| \leq 1$ where h_L and h_R are the heights of T_L and T_R , respectively.

【Definition, AVL tree】 The balance factor $BF(\text{ node }) = h_L - h_R$. In an AVL tree, $BF(\text{ any node }) = -1, 0, \text{ or } 1$.

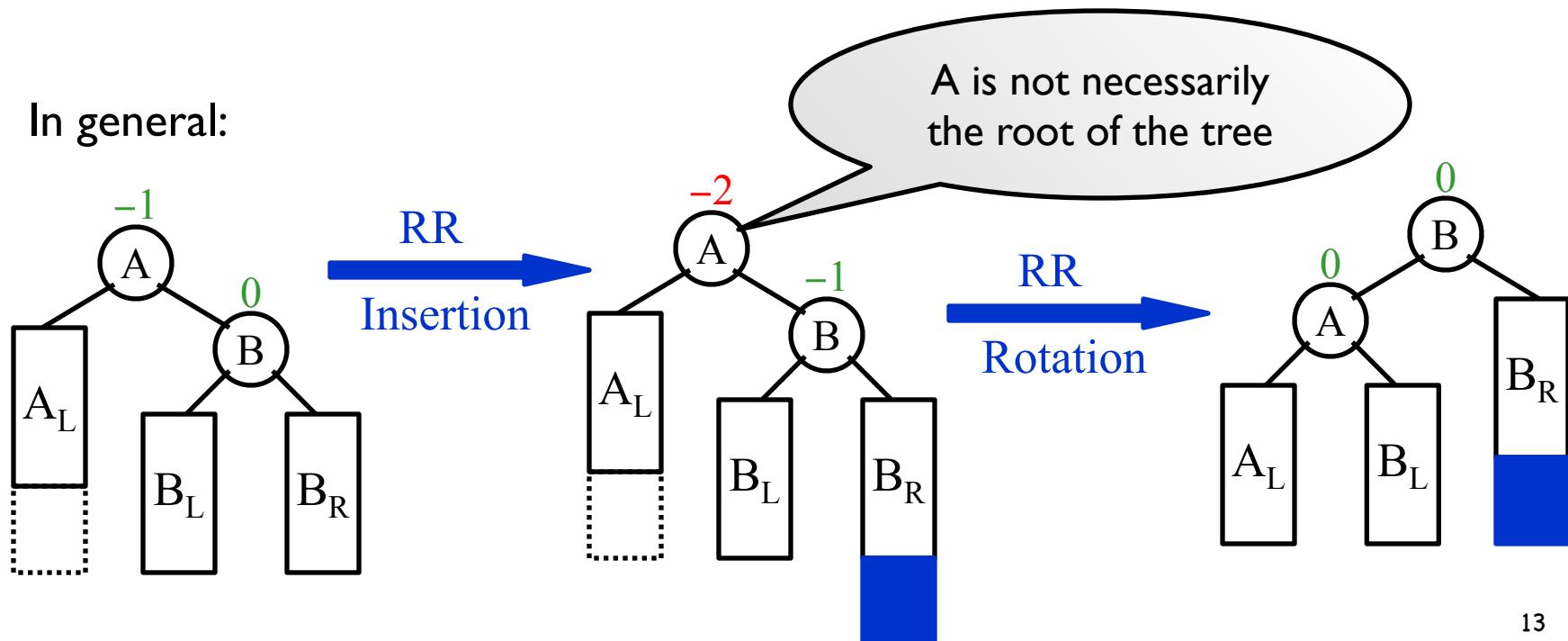


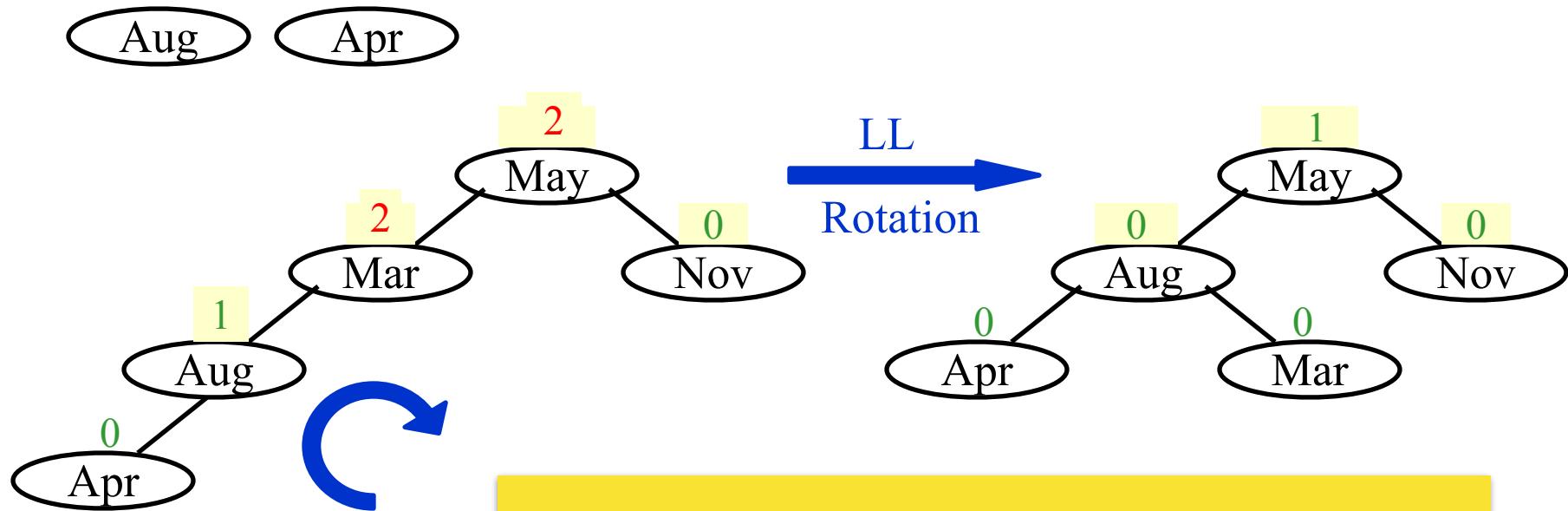
【Example】 Input the months



The trouble maker **Nov** is in the **right subtree's right subtree** of the trouble finder **Mar**. Hence it is called an **RR rotation**.

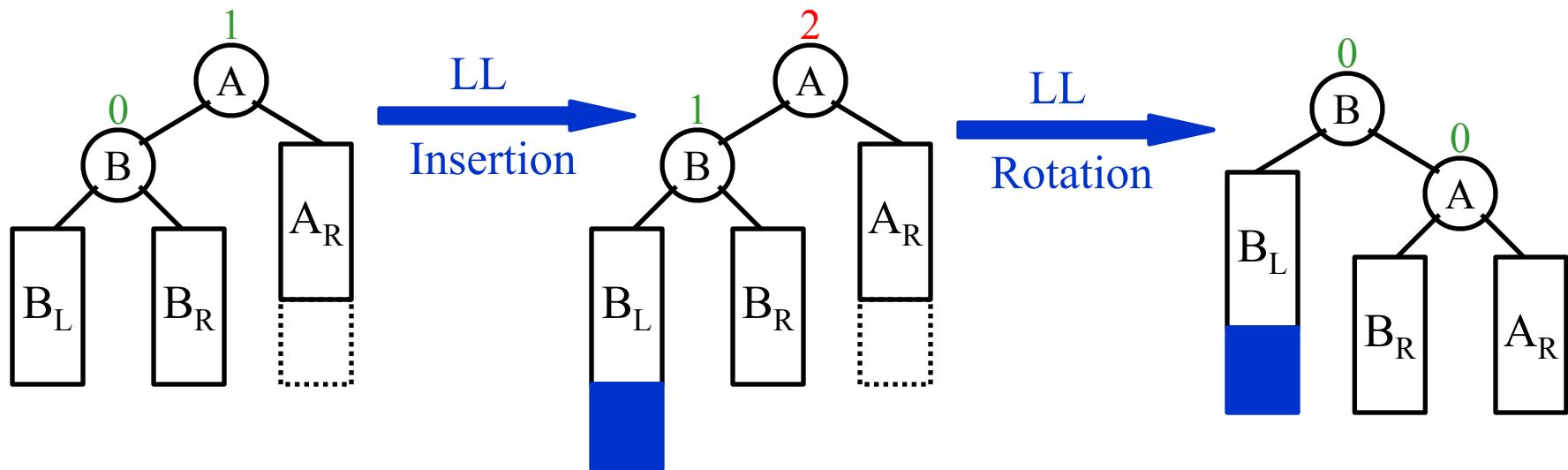
In general:

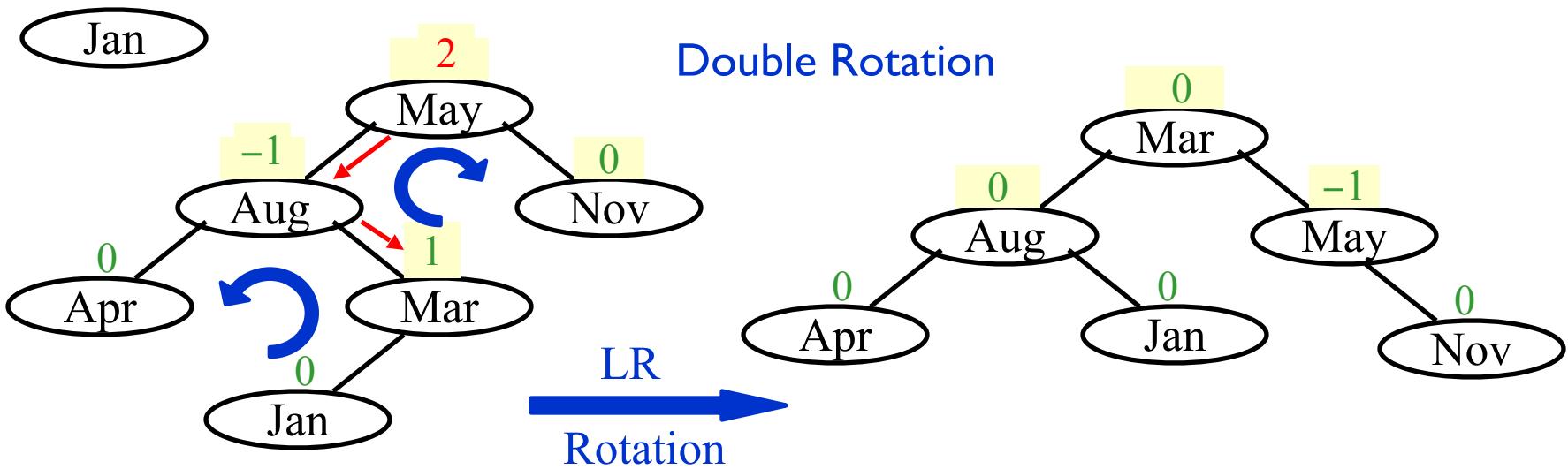




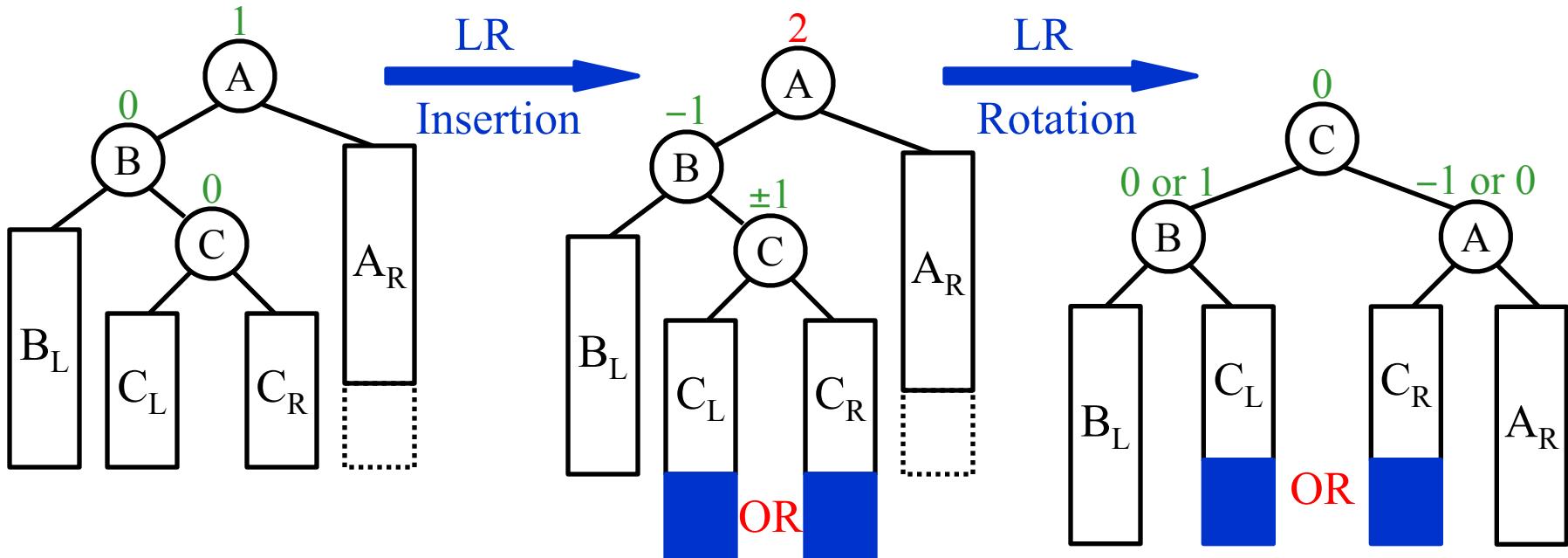
What can we do now?

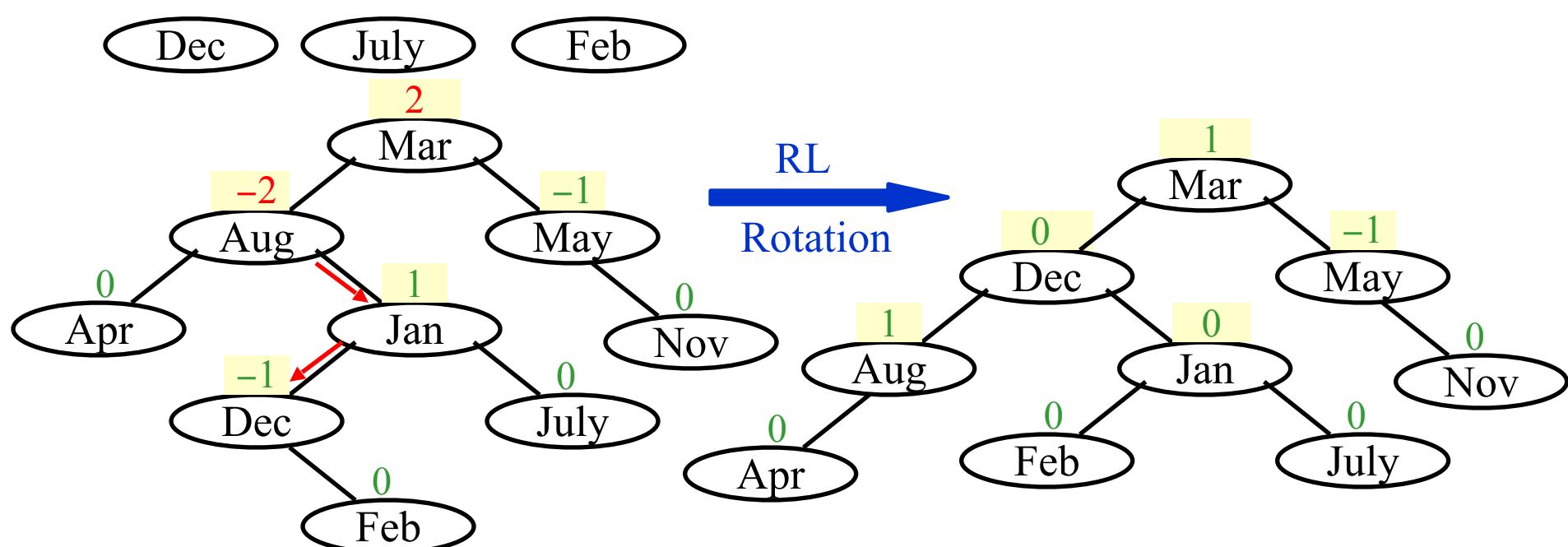
In general:



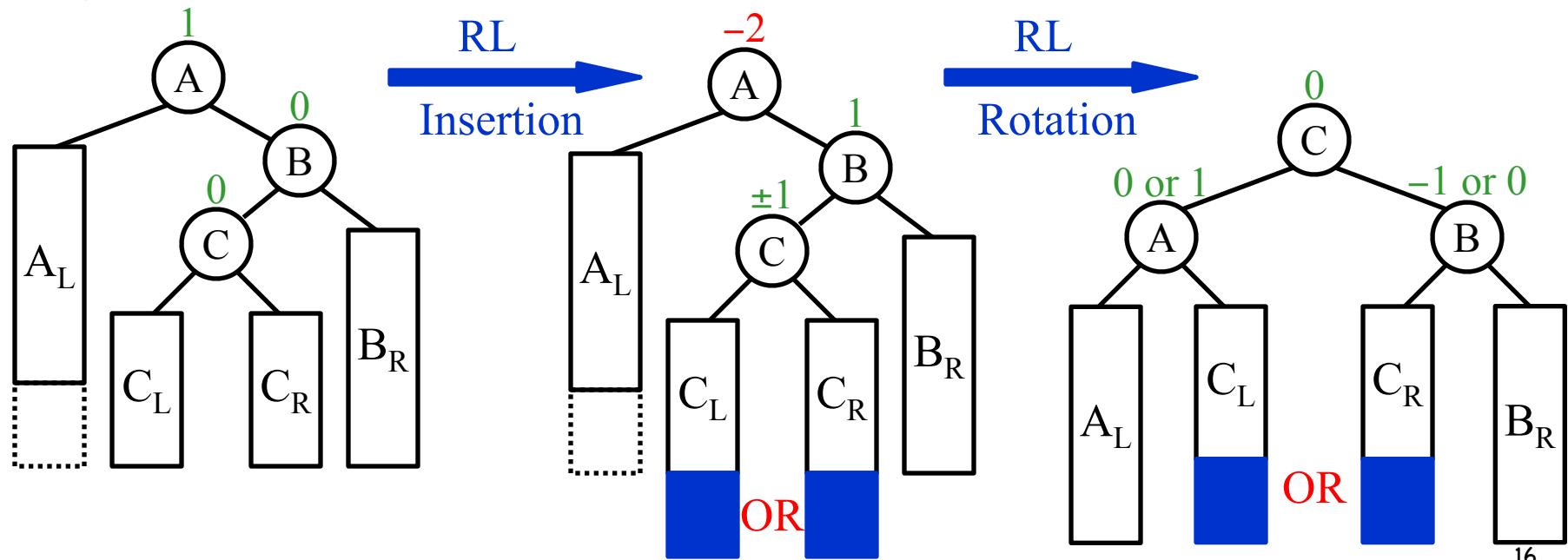


In general:





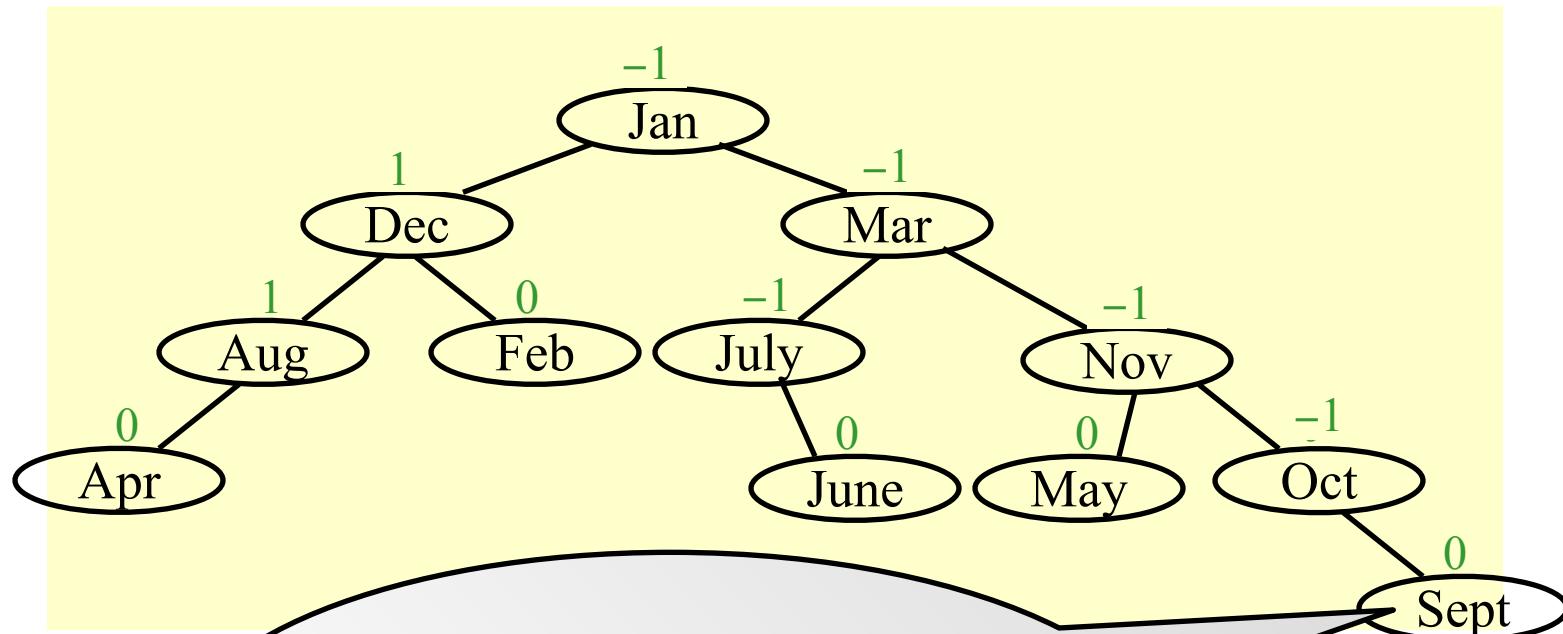
In general:



June

Oct

Sept

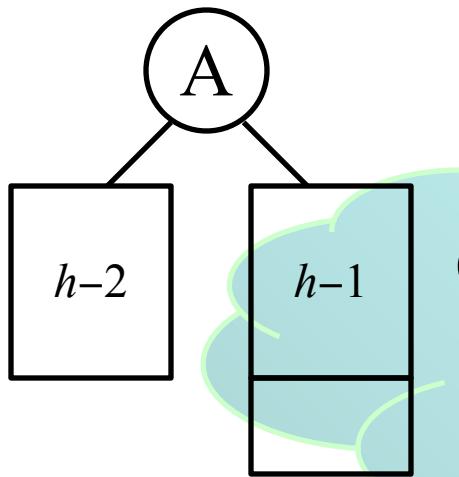


Note: Several bf's
might be changed even if
we don't need to reconstruct
the tree.

Another option is to keep a *height* field for each node.

Read the declaration and functions in [Weiss] Figures 4.42 – 4.48

Let n_h be the minimum number of nodes in a height-balanced tree of height h . What does the tree look like?



OR

If one last question:
Obviously we have $T_p = O(h)$
where h is the height of the tree.
But $h = ?$

The worst case for AVL tree of height h .

Fibonacci numbers:

$$F_0 = 0, F_1 = 1, F_i = F_{i-1} + F_{i-2} \text{ for } i > 1$$

$$\Rightarrow n_h = F_{h+3} - 1, \text{ for } h \geq 0$$

Fibonacci number theory gives that $F_i \approx \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^i$

$$\Rightarrow n_h \approx \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{h+3} - 1 \quad \Rightarrow \quad h = O(\ln n)$$



Outline: Balanced Binary Search Trees (I)

- Binary search trees
- AVL trees
- Splay trees
- Amortized analysis
- Take-home messages

Splay Trees (1985)



Daniel Sleator



Robert Tarjan

Self-Adjusting Binary Search Trees

DANIEL DOMINIC SLEATOR AND ROBERT ENDRE TARJAN

AT&T Bell Laboratories, Murray Hill, NJ

Figure courtesy: <https://csd.cmu.edu/people/faculty/daniel-sleator>
https://en.wikipedia.org/wiki/Robert_Tarjan

Splay Trees

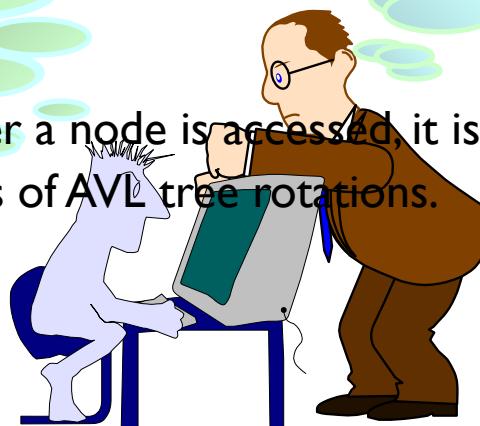


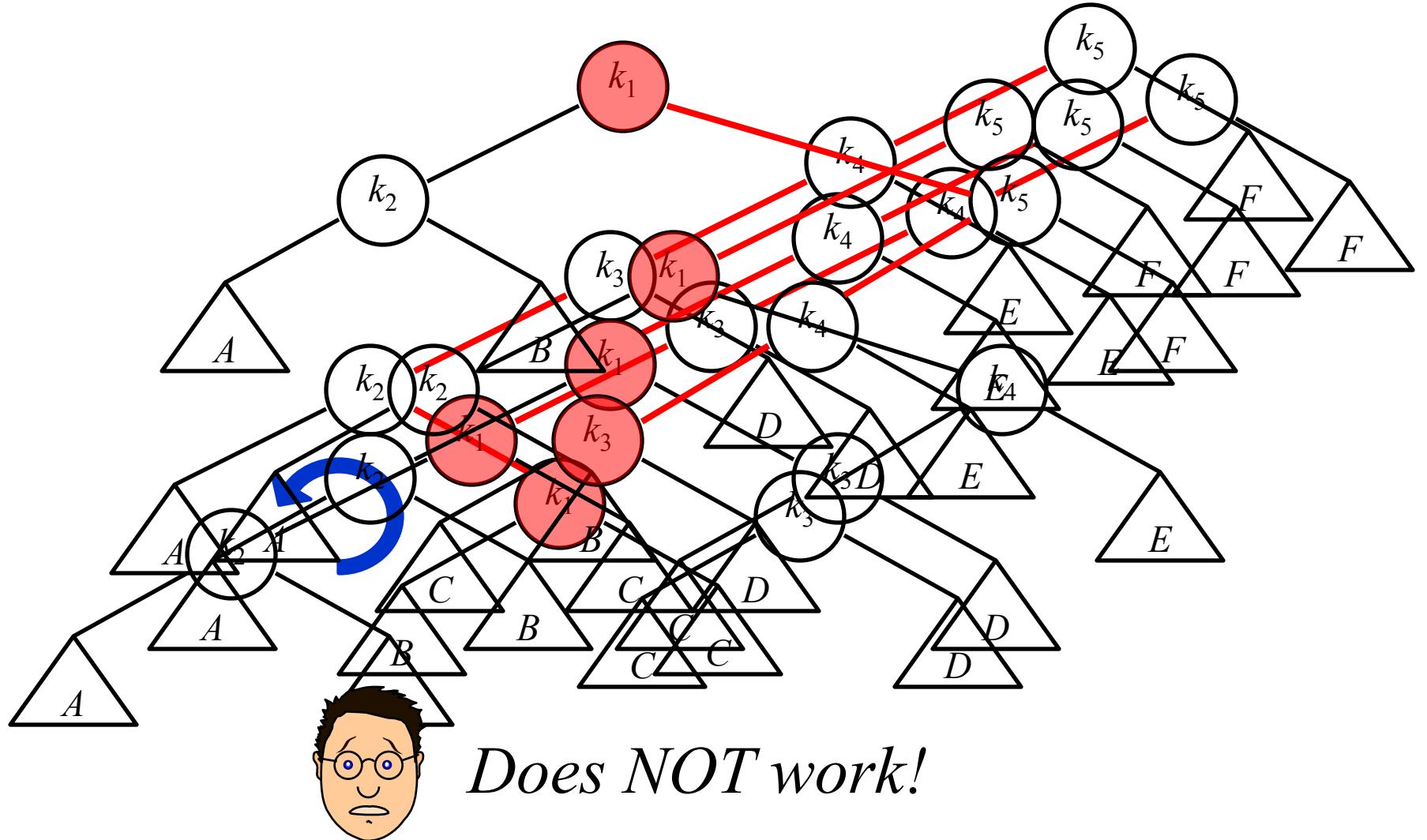
Target : Any M consecutive tree operations **starting from an empty tree** take at most $O(M \log N)$ time.

Surely we can – that only means that whenever a node is accessed, it must be **moved**. Otherwise visiting a bad node repeatedly leads to bad performance . If one node takes $O(N)$ time to access, we can keep accessing it for M times, can't we?



Idea : After a node is accessed, it is pushed to the root by a series of AVL tree rotations.

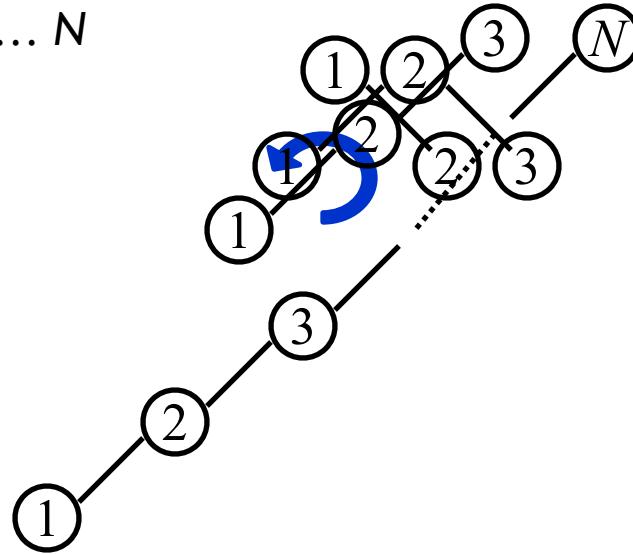




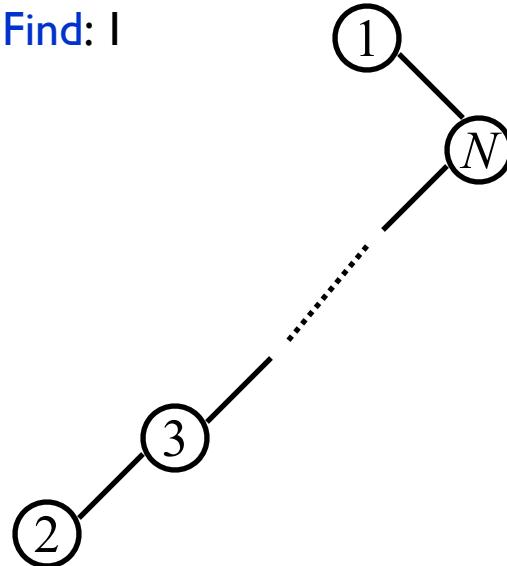
The rotation pushes other nodes deeper

An even worse case:

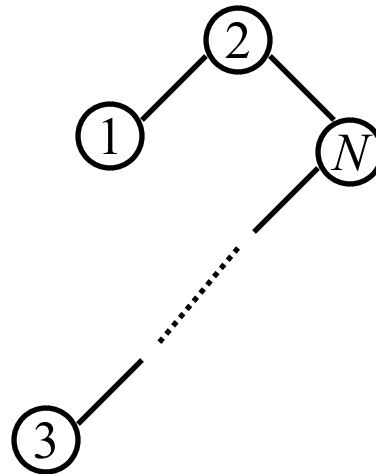
Insert: 1, 2, 3, ... N



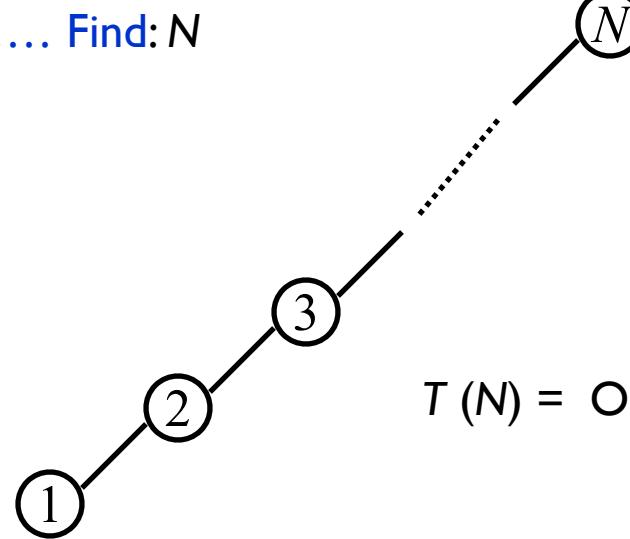
Find: 1



Find: 2



..... Find: N



$$T(N) = O(N^2)$$

Try again -- For any nonroot node X , denote its parent by P and grandparent by G :

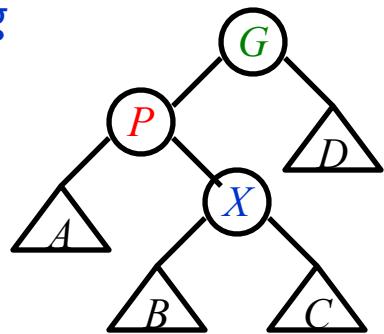
Zig

Case 1: P is the root

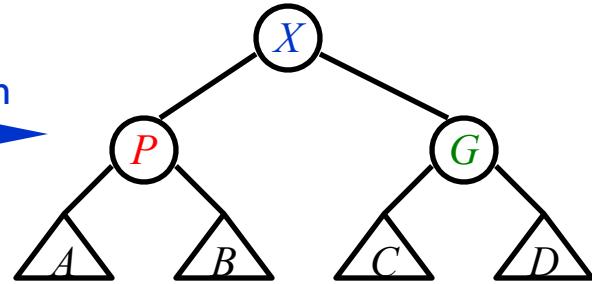
→ Rotate X and P

Case 2: P is not the root

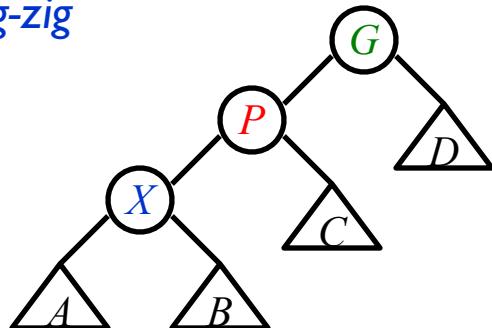
Zig-zag



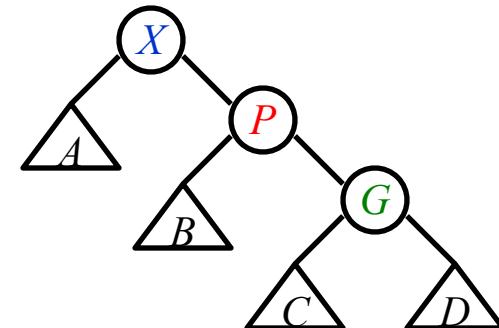
Double rotation



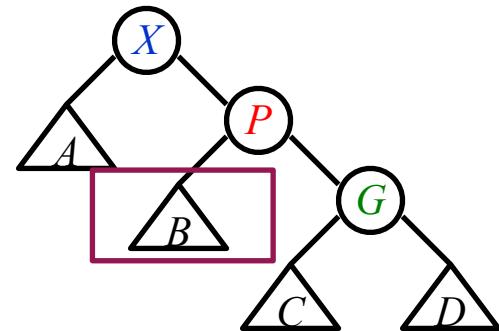
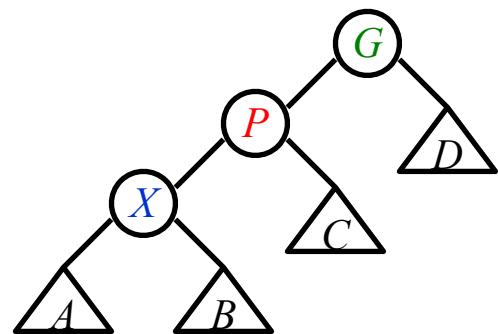
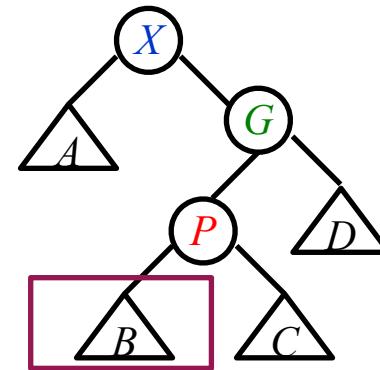
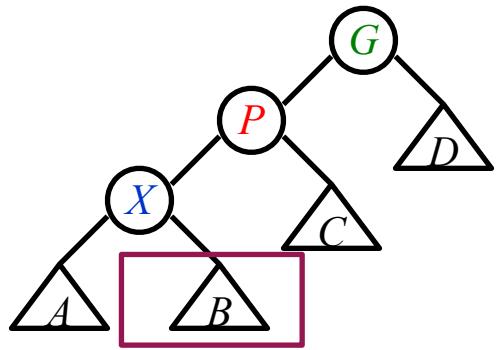
Zig-zig



Single rotation

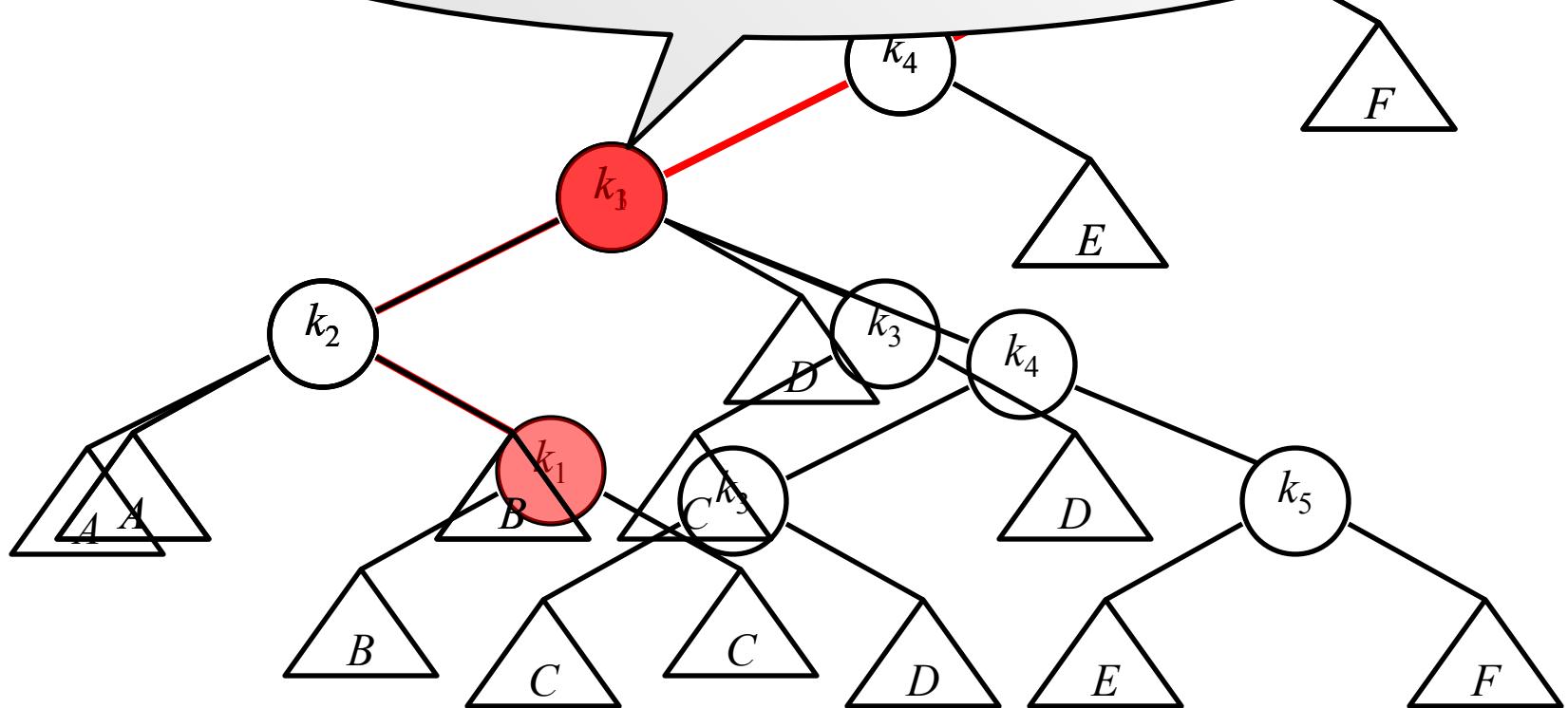


Compare for the Zig-zig case:



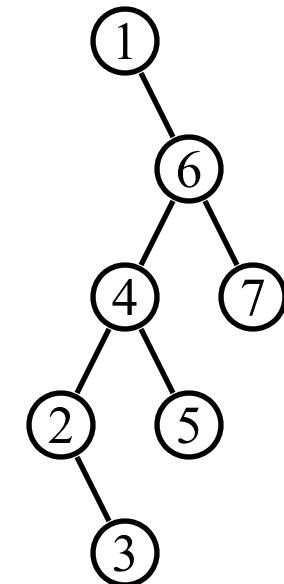
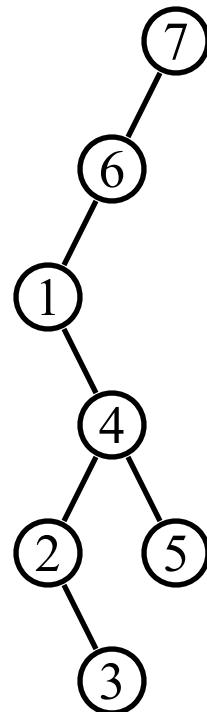
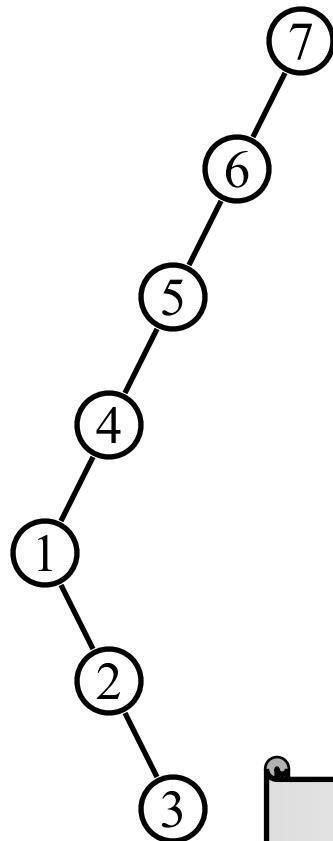
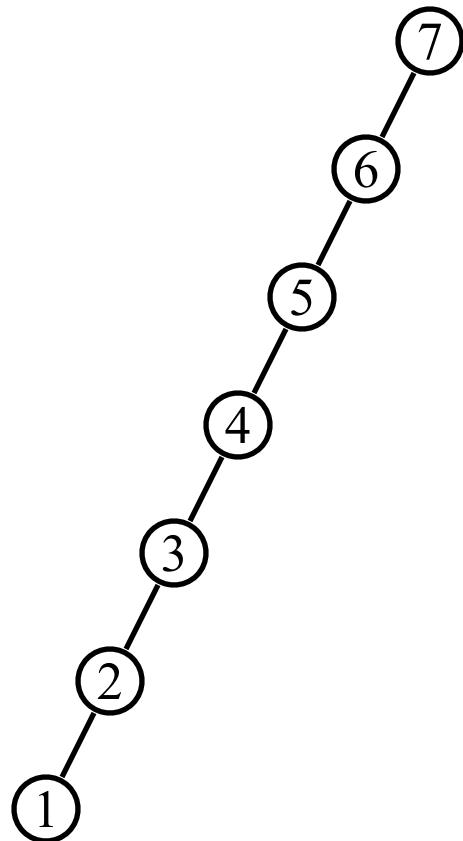
For zig-zig case, the right child of the node to splay avoids going deep

Splaying not only moves the accessed node to the root, but also roughly halves the depth of most nodes on the path.



Insert: 1, 2, 3, 4, 5, 6, 7

Find: 1



Read the 32-node example given
in [Weiss] Figures 4.52 – 4.60

Operations on Splay Trees

Deletions:

👉 Step 1: Find X ;

X will be at the root due to splaying.

👉 Step 2: Remove X ;

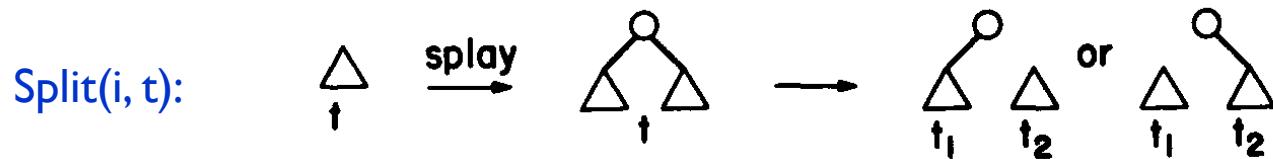
There will be two subtrees T_L and T_R .

👉 Step 3: FindMax (T_L) ;

The largest element will be the root of T_L , and *has no right child*.

👉 Step 4: Make T_R the right child of the root of T_L .

Operations on Splay Trees



All operations involve a series of splay steps.

Check the details in the “Self-adjusting binary search trees” paper.

Next, we study the complexity of splay tree operations.

Outline: Balanced Binary Search Trees (I)

- Binary search trees
- AVL trees
- Splay trees
- Amortized analysis
- Take-home messages

Amortized Analysis



Target : Any M consecutive operations take at most $O(M \log N)$ time.
-- *Amortized* time bound

In the sense that whether the bound holds for any input case

worst-case bound

stricter
than

amortized bound

stricter
than

average-case bound

Probability
is not involved



Aggregate analysis



Accounting method



Potential method

The Two-Stack Queue

Start from
empty

Out

In

The Two-Stack Queue

Start from
empty

Push



Out



In

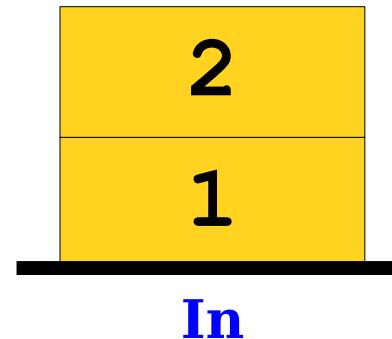
The Two-Stack Queue

Start from
empty

Push

Push

Out



The Two-Stack Queue

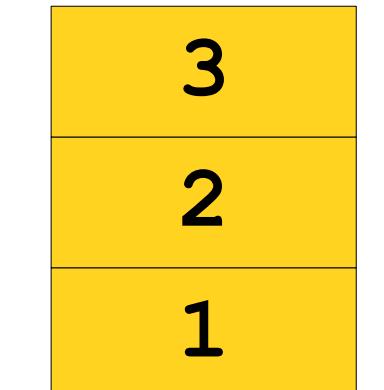
Start from
empty

Push

Push

Push

Out



The Two-Stack Queue

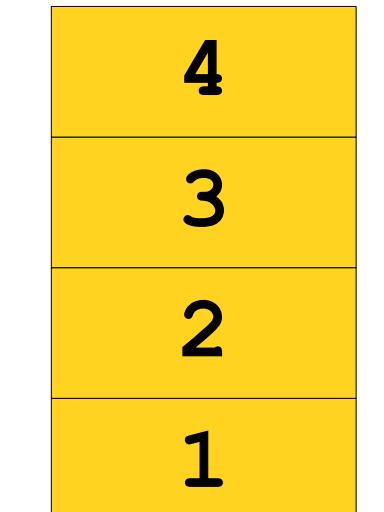
Start from
empty

Push

Push

Push

Push



The Two-Stack Queue

Start from
empty

Push

Push

Push

Push

Pop with
transfer

4

3

2

1

Out

In

The Two-Stack Queue

Start from
empty

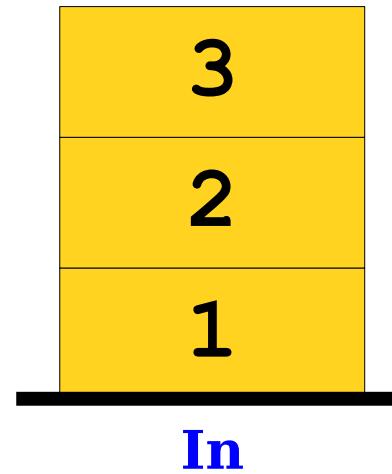
Push

Push

Push

Push

Pop with
transfer



The Two-Stack Queue

Start from
empty

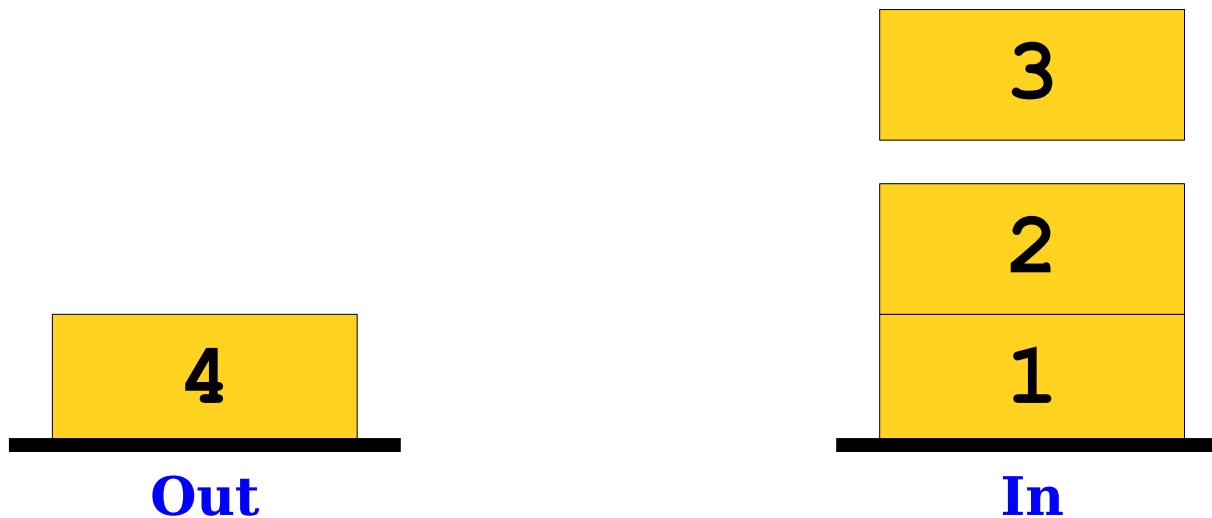
Push

Push

Push

Push

Pop with
transfer



The Two-Stack Queue

Start from
empty

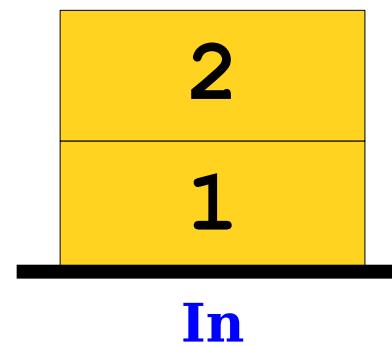
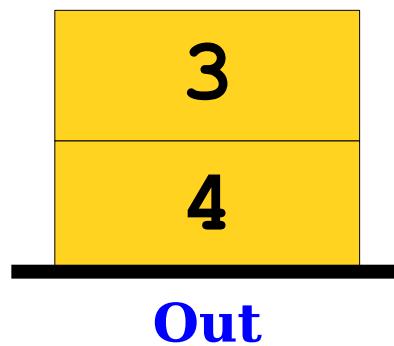
Push

Push

Push

Push

Pop with
transfer



The Two-Stack Queue

Start from
empty

Push

Push

Push

Push

Pop with
transfer

2

3

4

Out

1

In

The Two-Stack Queue

Start from
empty

Push

Push

Push

Push

Pop with
transfer



The Two-Stack Queue

Start from
empty

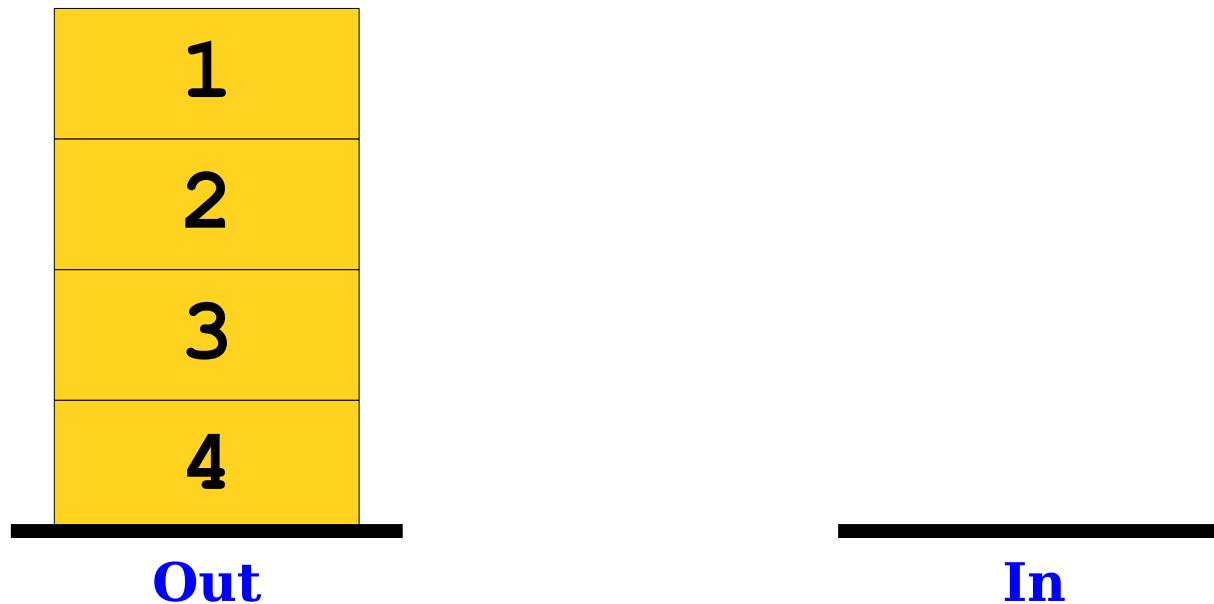
Push

Push

Push

Push

Pop with
transfer



The Two-Stack Queue

Start from
empty

Push

Push

Push

Push

Pop with
transfer



1

The Two-Stack Queue

Start from
empty

Push

Push

Push

Push

Pop with
transfer

Pop



1

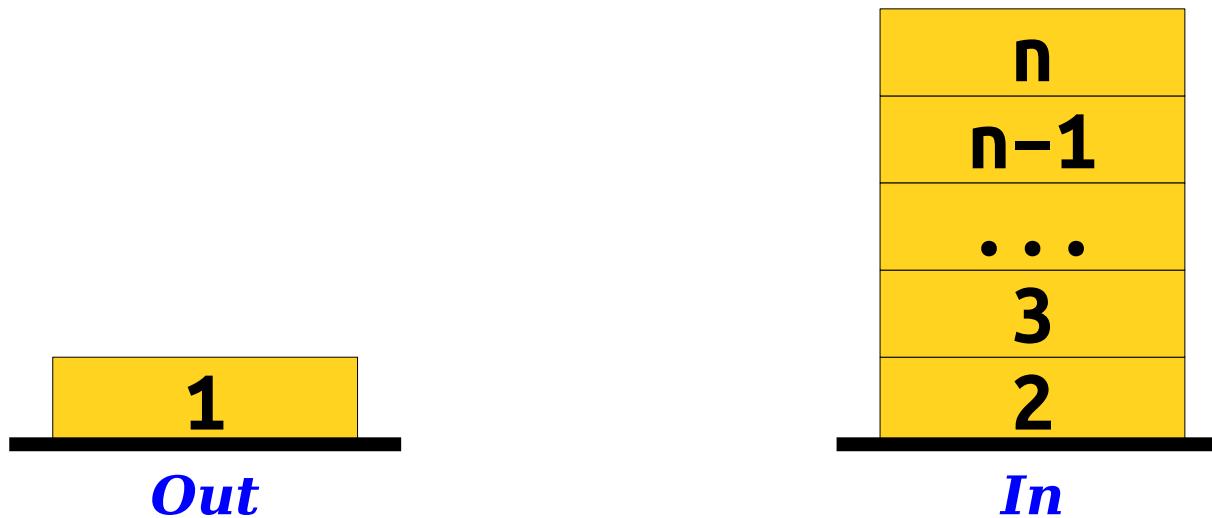
2

The Two-Stack Queue

- Maintain an ***In*** stack and an ***Out*** stack.
- To enqueue an element, push it onto the ***In*** stack.
- To dequeue an element:
 - If the ***Out*** stack is nonempty, pop it.
 - If the ***Out*** stack is empty, pop elements from the ***In*** stack, pushing them into the ***Out*** stack. Then dequeue as usual.

The Two-Stack Queue

- Each enqueue takes time $O(1)$.
 - Just push an item onto the ***In*** stack.
- Dequeues can vary in their runtime.
 - Could be $O(1)$ if the ***Out*** stack isn't empty.
 - Could be $\Theta(n)$ if the ***Out*** stack is empty.



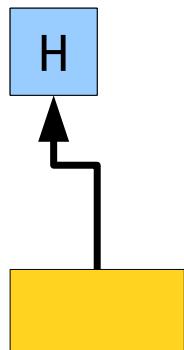
The Two-Stack Queue

- **Key Fact:** Any series of n operations on an (initially empty) two-stack queue will take time $O(n)$.
- **Why?**
 - Each item is pushed into at most two stacks and popped from at most two stacks.
 - Adding up the work done per element across all n operations, we can do at most $O(n)$ work.



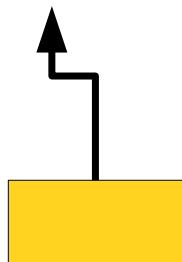
Dynamic Arrays

- A ***dynamic array*** is the most common way to implement a list of values.
- Maintain an array slightly bigger than the one you need. When you run out of space, double the array size and copy the elements over.



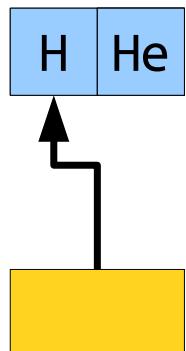
Dynamic Arrays

- A ***dynamic array*** is the most common way to implement a list of values.
- Maintain an array slightly bigger than the one you need. When you run out of space, double the array size and copy the elements over.



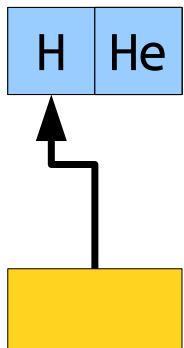
Dynamic Arrays

- A ***dynamic array*** is the most common way to implement a list of values.
- Maintain an array slightly bigger than the one you need. When you run out of space, double the array size and copy the elements over.



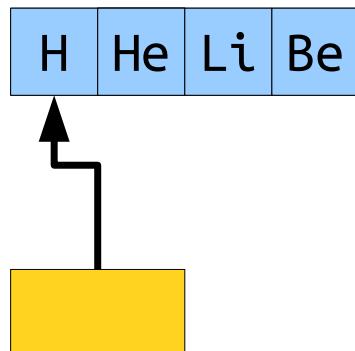
Dynamic Arrays

- A ***dynamic array*** is the most common way to implement a list of values.
- Maintain an array slightly bigger than the one you need. When you run out of space, double the array size and copy the elements over.



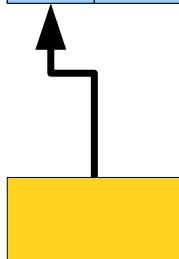
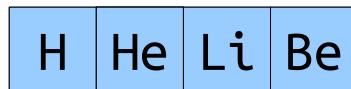
Dynamic Arrays

- A ***dynamic array*** is the most common way to implement a list of values.
- Maintain an array slightly bigger than the one you need. When you run out of space, double the array size and copy the elements over.



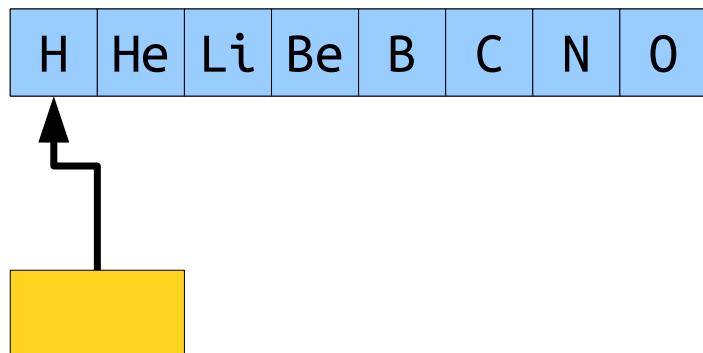
Dynamic Arrays

- A ***dynamic array*** is the most common way to implement a list of values.
- Maintain an array slightly bigger than the one you need. When you run out of space, double the array size and copy the elements over.



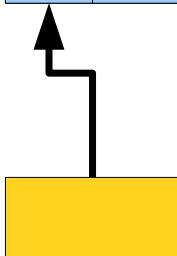
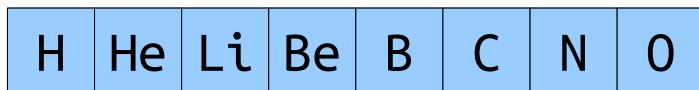
Dynamic Arrays

- A ***dynamic array*** is the most common way to implement a list of values.
- Maintain an array slightly bigger than the one you need. When you run out of space, double the array size and copy the elements over.



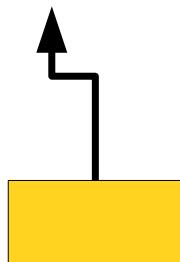
Dynamic Arrays

- A ***dynamic array*** is the most common way to implement a list of values.
- Maintain an array slightly bigger than the one you need. When you run out of space, double the array size and copy the elements over.



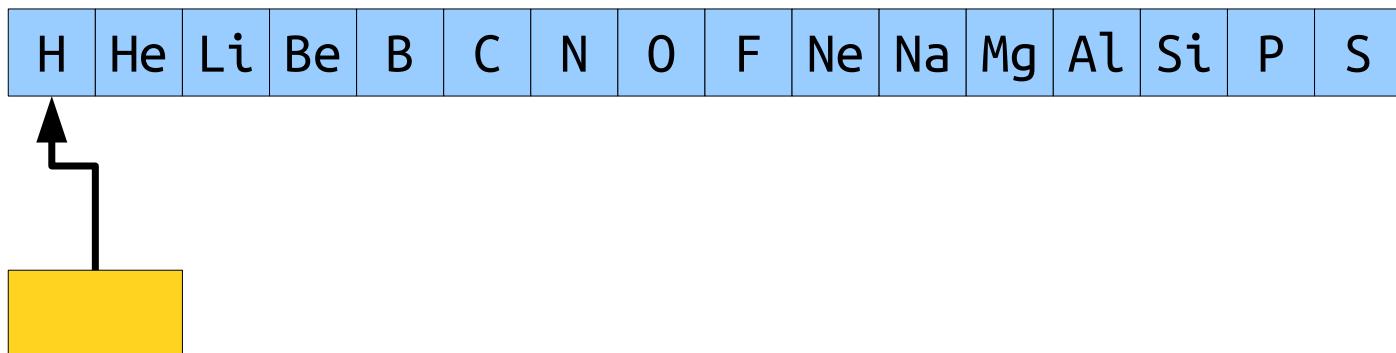
Dynamic Arrays

- A ***dynamic array*** is the most common way to implement a list of values.
- Maintain an array slightly bigger than the one you need. When you run out of space, double the array size and copy the elements over.



Dynamic Arrays

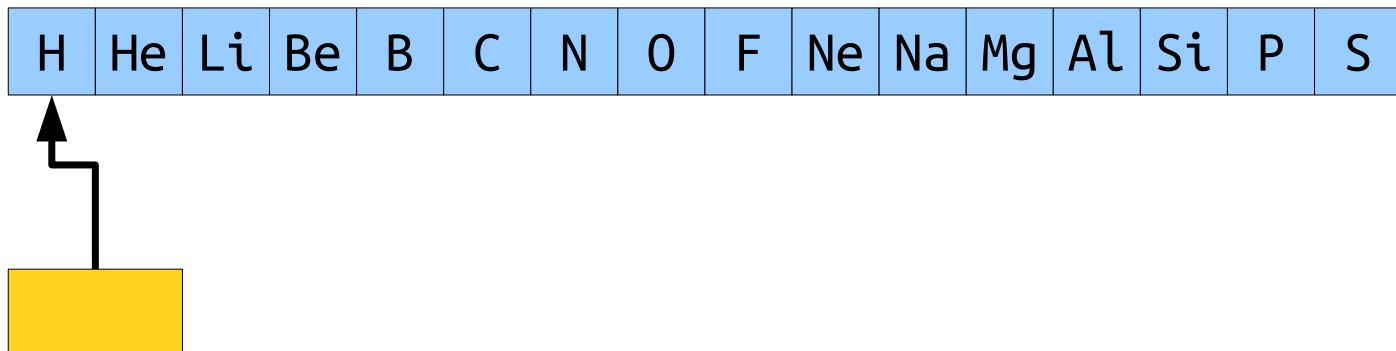
- A ***dynamic array*** is the most common way to implement a list of values.
- Maintain an array slightly bigger than the one you need. When you run out of space, double the array size and copy the elements over.



Dynamic Arrays

- **Claim:** Appending n elements always takes time $O(n)$.
- The array doubles at sizes $2^0, 2^1, 2^2, \dots$, etc.
- The very last doubling is at the largest power of two less than n . This is at most $2^{\lfloor \log_2 n \rfloor}$. (Do you see why?)
- Total work done across all doubling is at most

$$\begin{aligned} 2^0 + 2^1 + \dots + 2^{\lfloor \log_2 n \rfloor} &= 2^{\lfloor \log_2 n \rfloor + 1} - 1 \\ &\leq 2^{\log_2 n + 1} \\ &= 2n. \end{aligned}$$



Aggregate Method



Idea : Show that for all n , a sequence of n operations takes **worst-case** time $T(n)$ in total. In the worst case, the average cost, per operation is therefore $T(n)/n$.

We can **pop** each object from the stack **at most once** for each time we have **pushed** it onto the stack

Total = $O(n^2)$?

【Example】 Stack with **Push**, **Pop**, **MultiPop**

Stack S)

```
Algorithm {  
    while ( !IsEmpty(S) && k>0 ) {  
        Pop(S);  
        k - -;  
    } /* end while-loop */  
}  
T = min ( sizeof(S), k )
```

Consider a sequence of **Push**, **Pop**, and **MultiPop** operations on an initially empty stack.

$$\text{sizeof}(S) \leq n$$

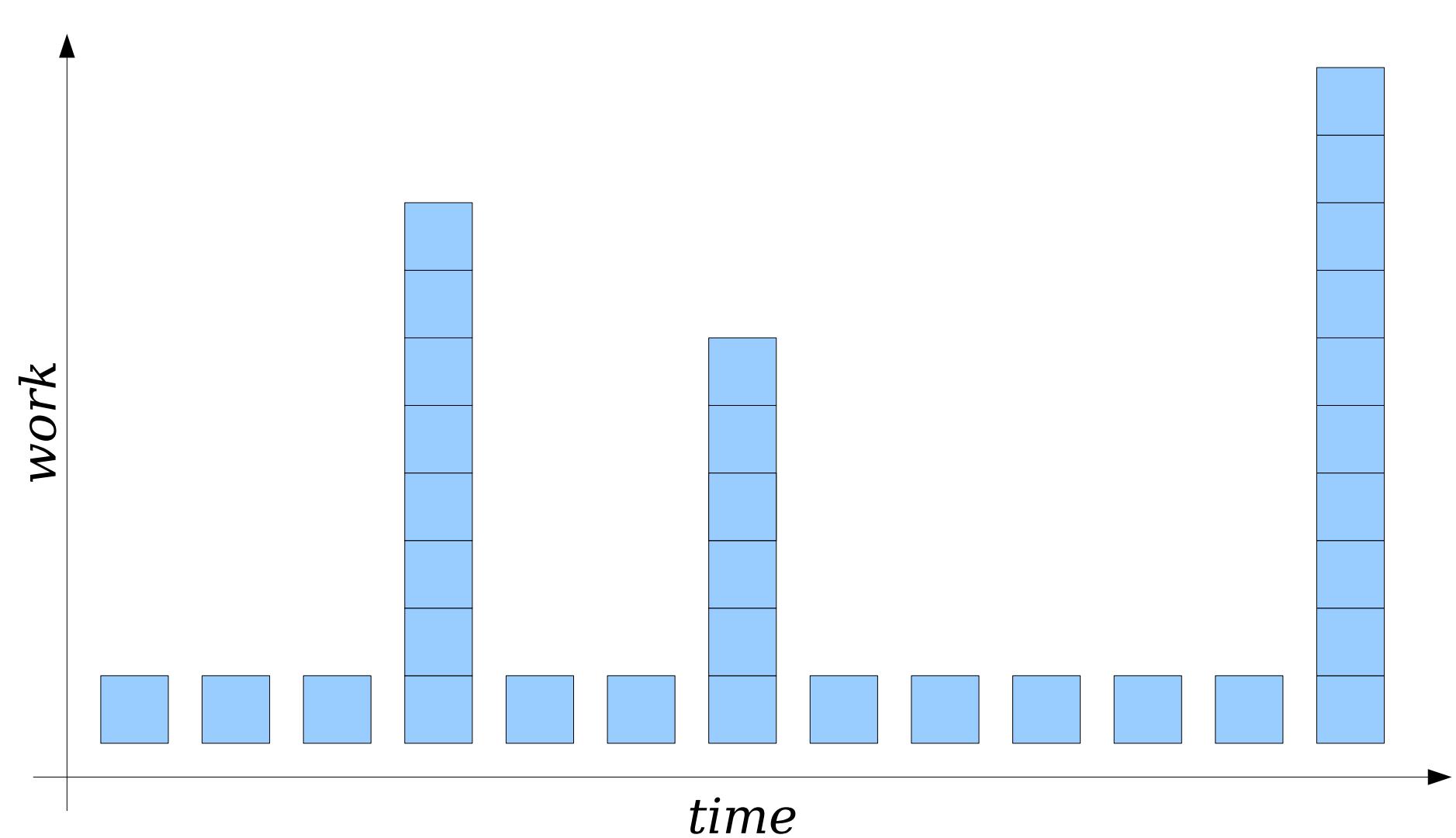
$$T_{\text{amortized}} = O(n)/n = O(1)$$

The total time of pop should be less than the total time of push.

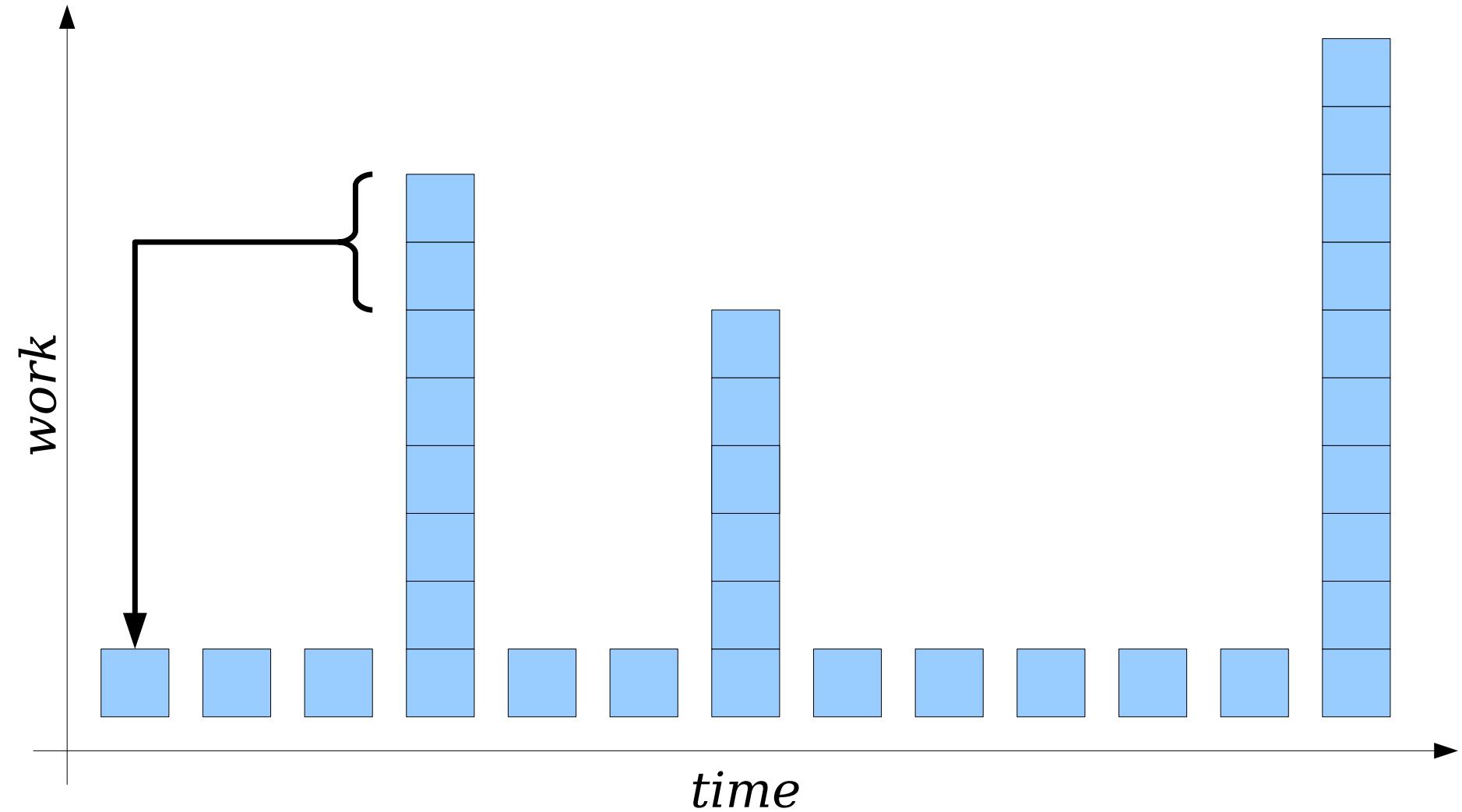
The total time of push takes at most $O(n)$.

Assigning Amortized Costs

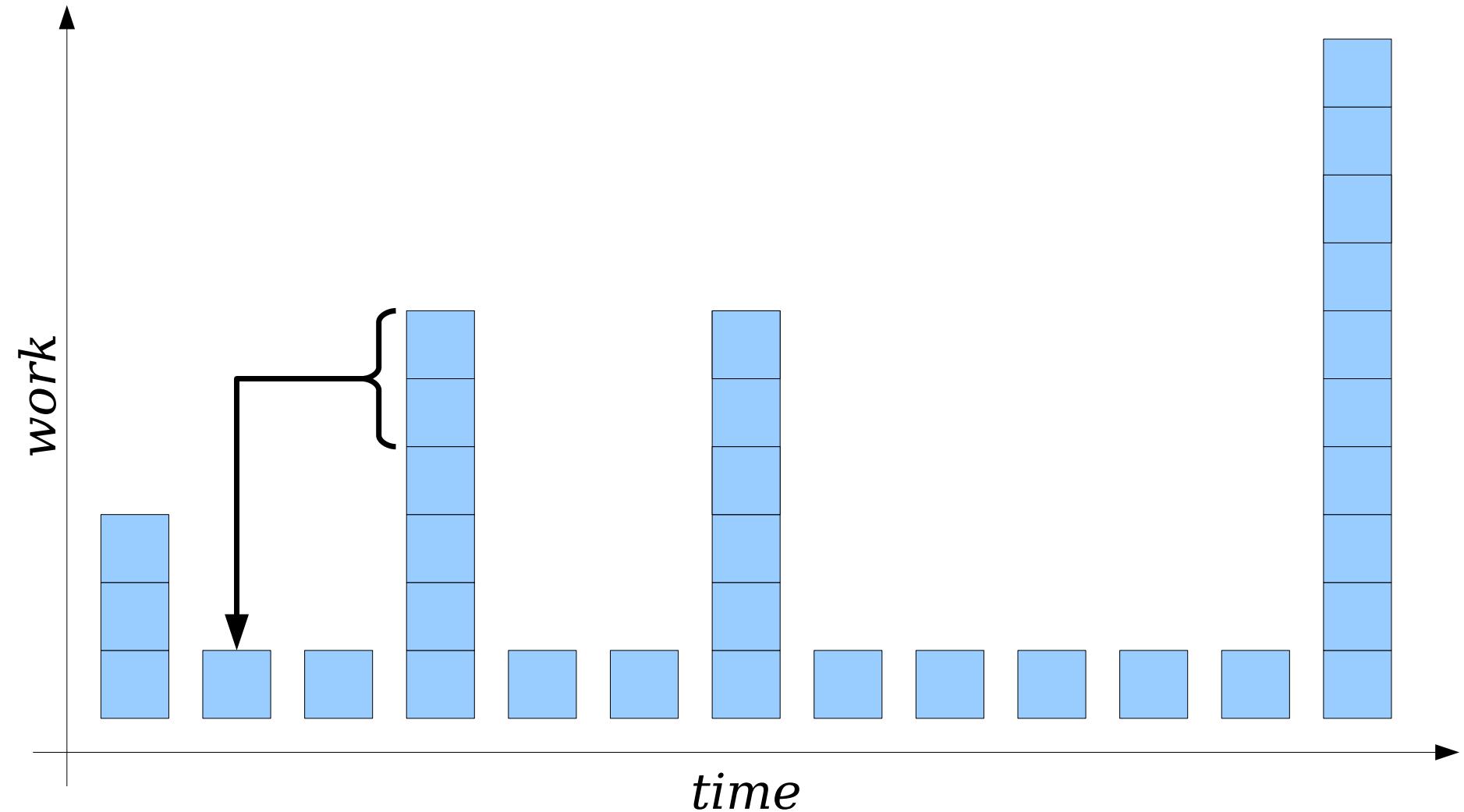
- The approach we've taken so far for assigning amortized costs is called an ***aggregate analysis***.
 - Directly compute the maximum possible work done across any sequence of operations, then divide that by the number of operations.
- This approach works well here, but it doesn't scale well to more complex data structures.
 - What if different operations contribute to / clean up messes in different ways?
 - What if it's not clear what sequence is the worst-case sequence of operations?
- In practice, we tend to use a different strategy called the ***potential method*** to assign amortized costs.



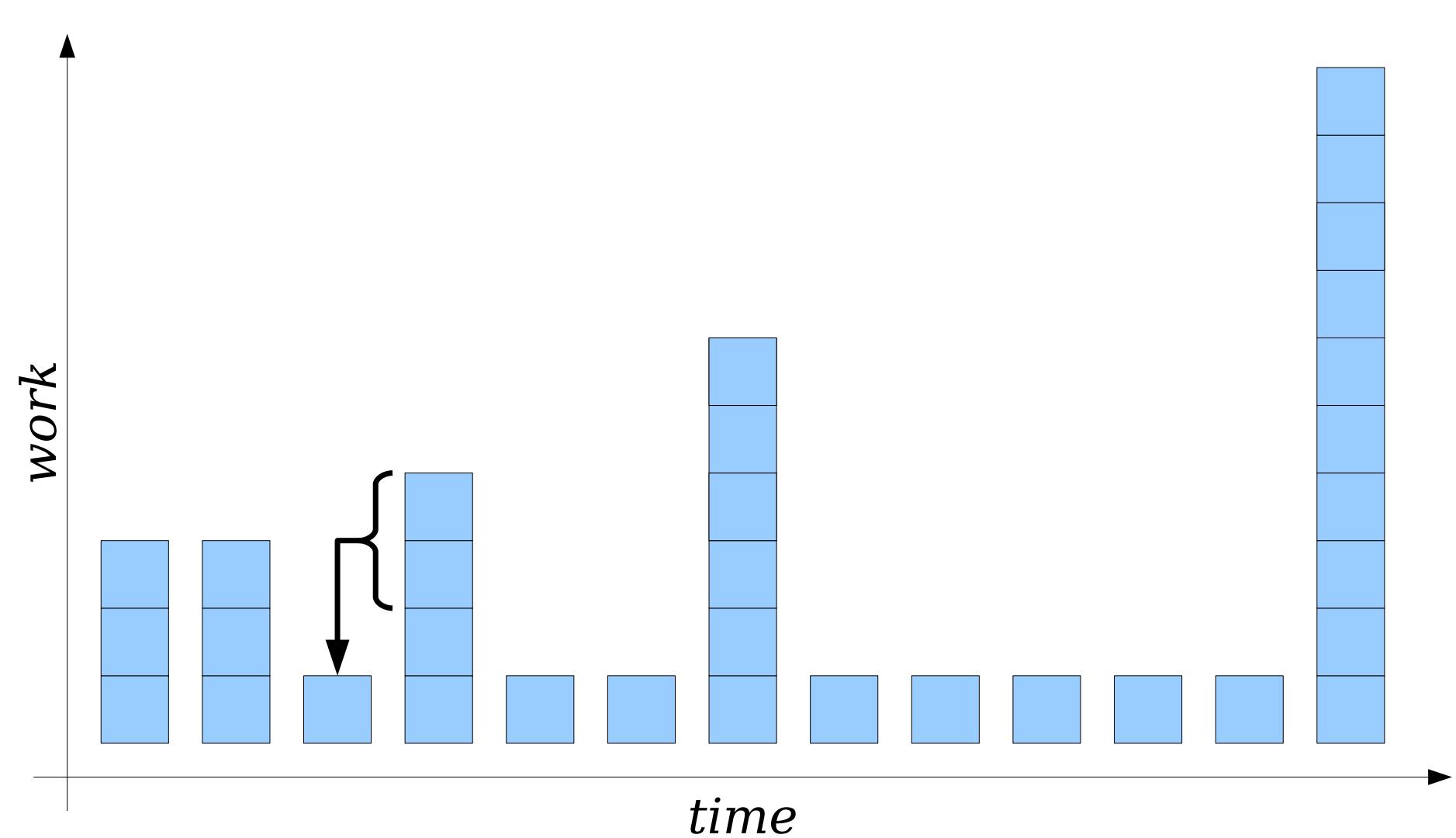
Key Idea: Backcharge expensive operations to cheaper ones.



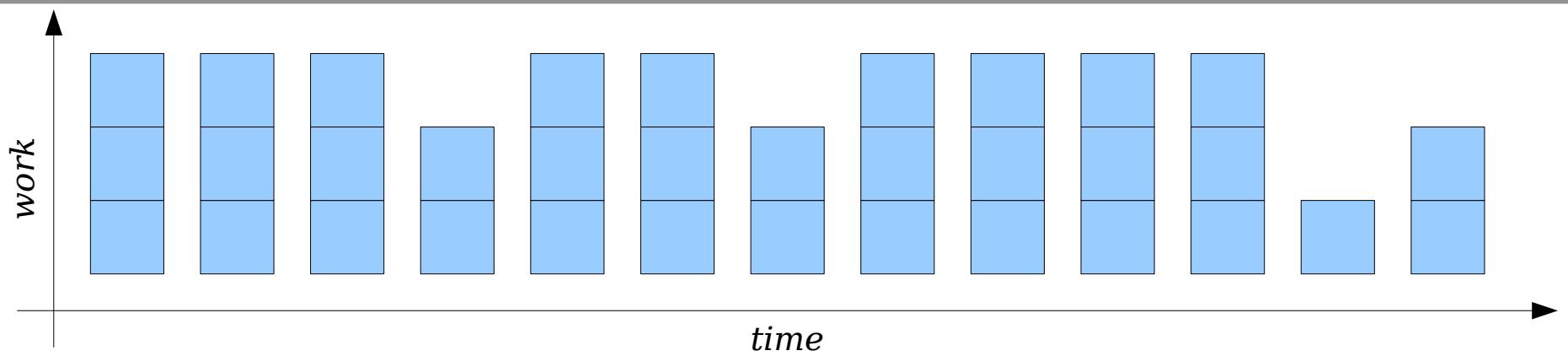
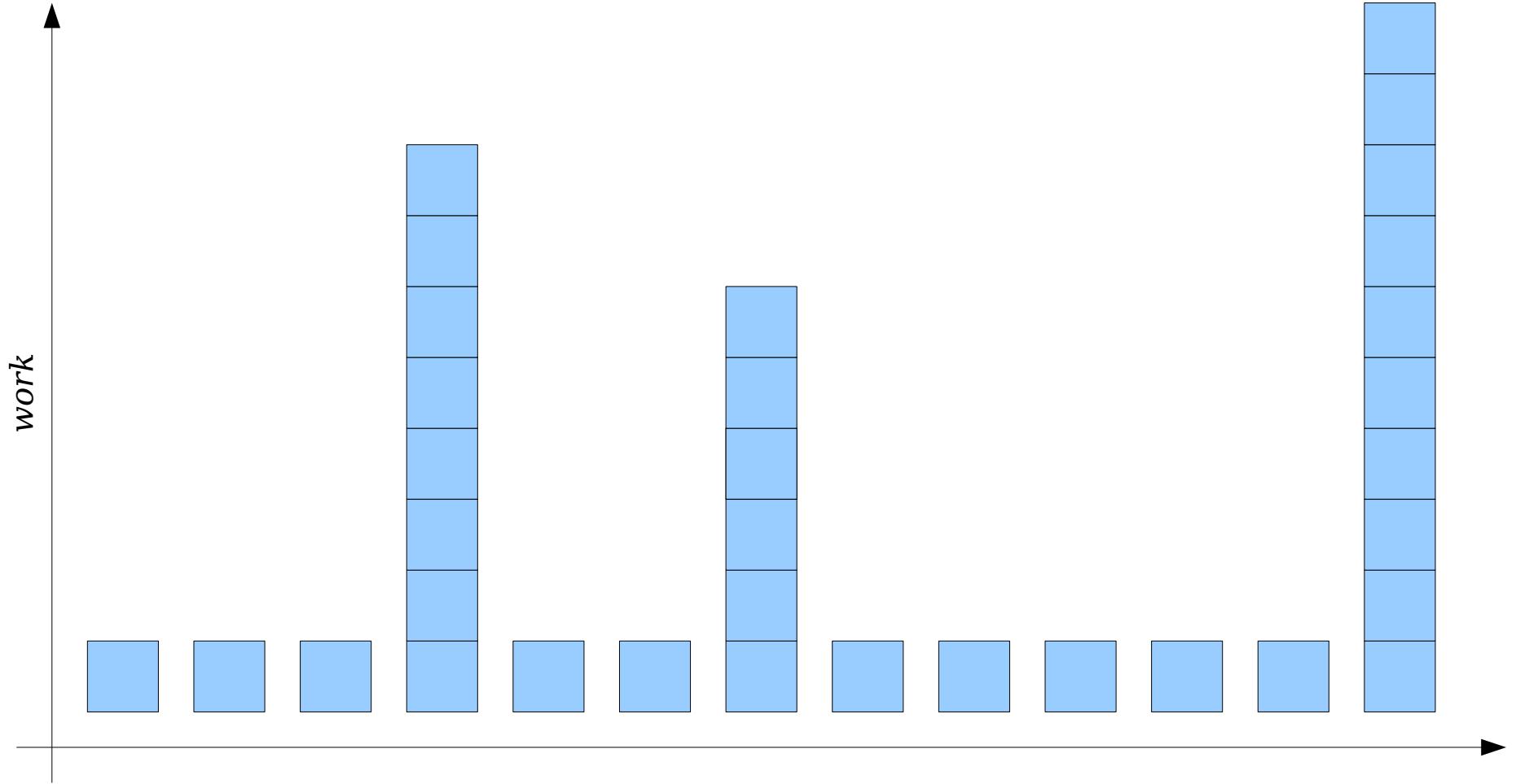
Key Idea: Backcharge expensive operations to cheaper ones.

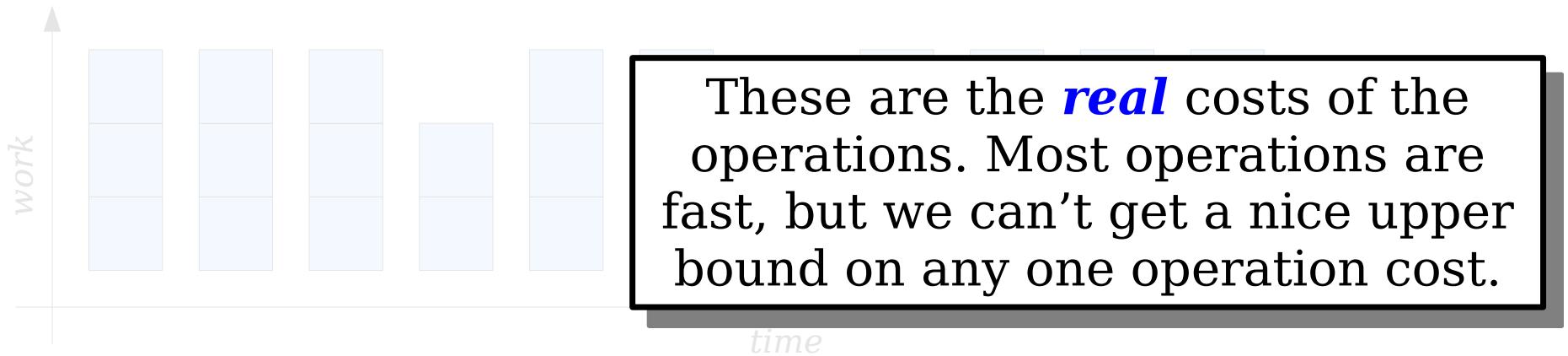
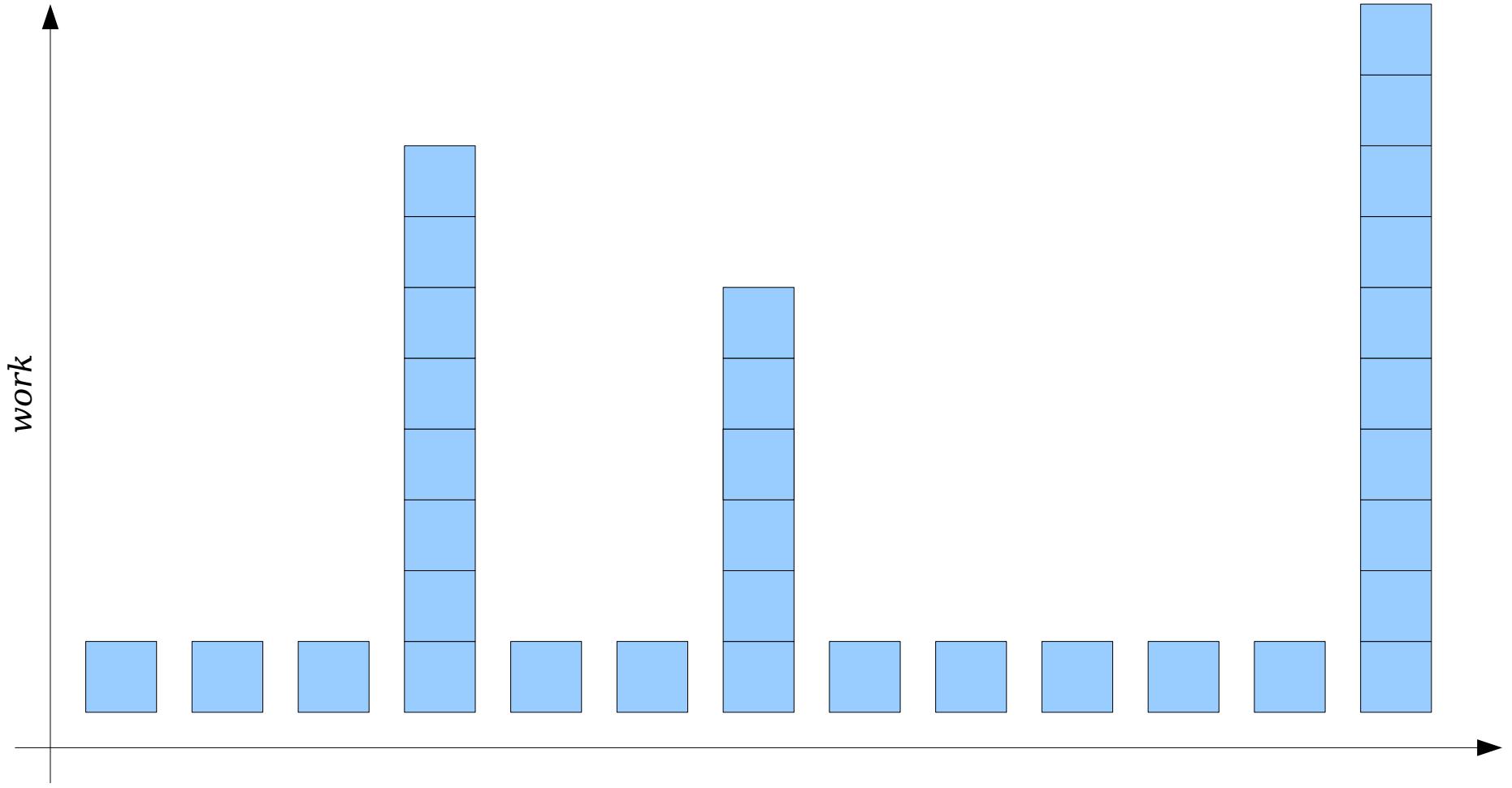


Key Idea: Backcharge expensive operations to cheaper ones.



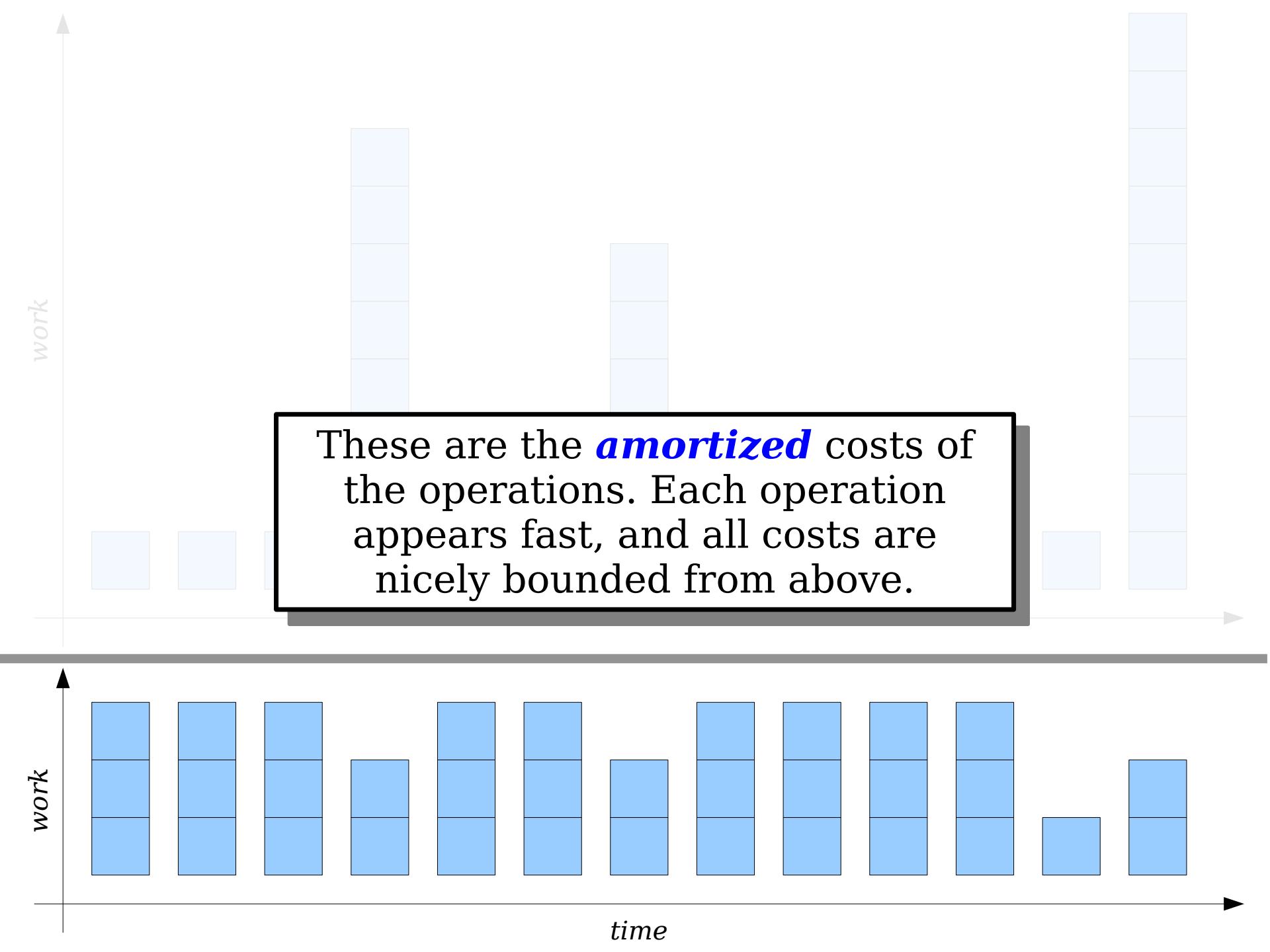
Key Idea: Backcharge expensive operations to cheaper ones.





These are the *real* costs of the operations. Most operations are fast, but we can't get a nice upper bound on any one operation cost.

work



These are the **amortized** costs of the operations. Each operation appears fast, and all costs are nicely bounded from above.

work

time

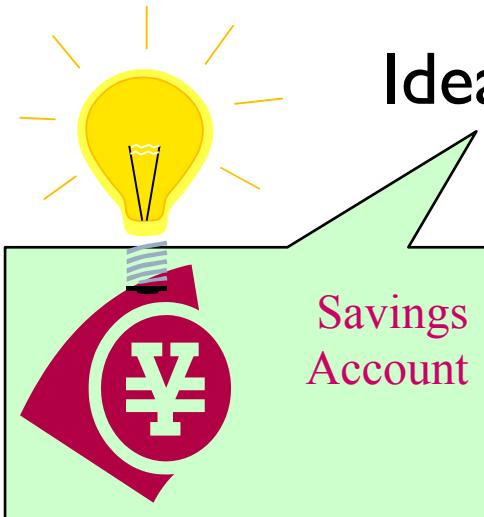
Amortized Analysis

- **Key Idea:** Assign each operation a (fake!) cost called its **amortized cost** such that, *for any series of operations performed*, the following is true:

$$\sum \text{amortized-cost} \geq \sum \text{real-cost}$$

- Amortized costs shift work backwards from expensive operations onto cheaper ones.
 - Cheap operations are artificially made more expensive to pay for future cleanup work.
 - Expensive operations are artificially made cheaper by shifting the work backwards.

Accounting Method



Idea : When an operation's *amortized cost* \hat{c}_i exceeds its *actual cost* c_i , we assign the difference to specific objects in the data structure as *credit*. Credit can help *pay* for later operations whose amortized cost is less than their actual cost.

Note: For all sequences of n operations, we must have

$$\sum_{i=1}^n \hat{c}_i \geq \sum_{i=1}^n c_i$$

$$T_{\text{amortized}} = \frac{1}{n} \sum_i \hat{c}_i$$

【Example】 Stack with **MultiPop**(int k, Stack S)

c_i for Push: 1 ; Pop: 1 ; and MultiPop: $\min(\text{sizeof}(S), k)$

\hat{c}_i for Push: 2 ; Pop: 0 ; and MultiPop: 0

Starting from an empty stack

Push: +1 ; Pop: -1

$\text{sizeof}(S) \geq 0$

$$\sum (\hat{c}_i - c_i)$$

The amortized costs of the operations may differ from each other

$$\rightarrow O(n) = \sum_{i=1}^n c_i \geq \sum_{i=1}^n \hat{c}_i$$

$$\rightarrow T_{\text{amortized}} = O(n)/n = O(1)$$

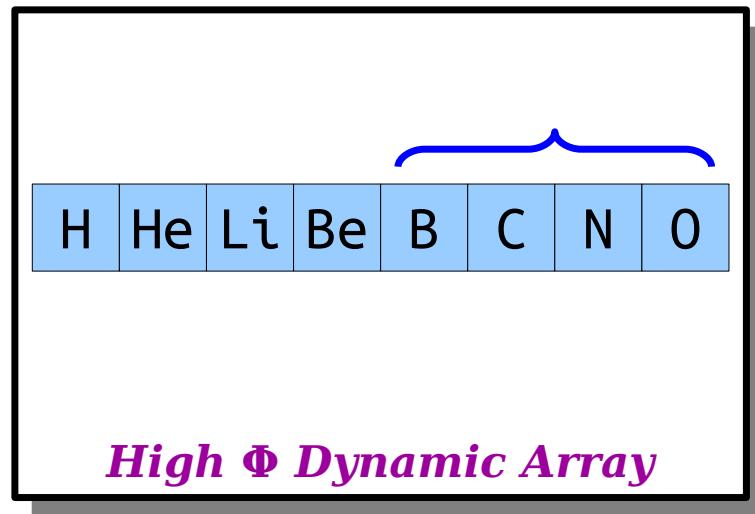
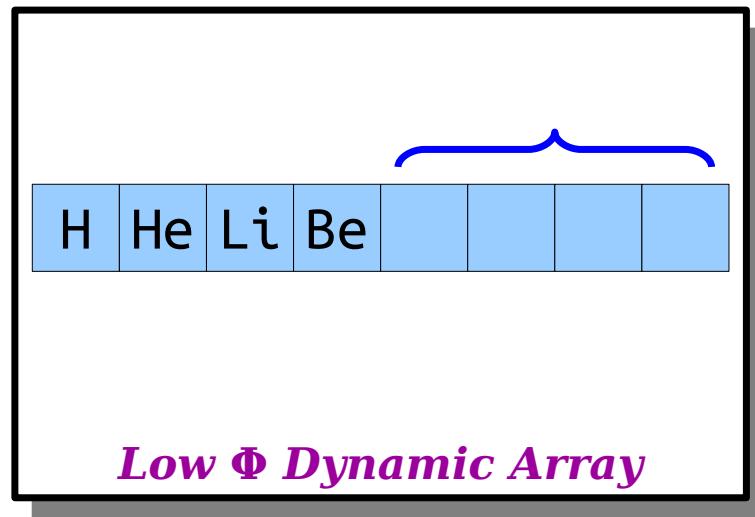
Potential Method

- Why some problems have smaller amortized time cost?
 - The structure of the problem provides the constraints:

All operations can not exceed the structural constraints.
- Represent the states of the structure as potential functions.
 - The potential function is bounded by the structural constraints.
 - Bound the total cost by the increase of potential.

Potential Functions

- To assign amortized costs, we'll need to measure how “messy” the data structure is.



Potential Functions

- Define the amortized cost of an operation to be

$$\text{amortized-cost} = \text{real-cost} + k \cdot \Delta\Phi$$

where k is a constant under our control and $\Delta\Phi$ is the difference between Φ just after the operation finishes and Φ just before the operation started:

$$\Delta\Phi = \Phi_{\text{after}} - \Phi_{\text{before}}$$

- Intuitively:
 - If Φ **increases**, the data structure got “**messier**,” and the amortized cost is **higher** than the real cost to account for future cleanup costs.
 - If Φ **decreases**, the data structure got “**cleaner**,” and the amortized cost is **lower** than the real cost

The Two-Stack Queue

Φ = height of **In** stack

Out

In

The Two-Stack Queue

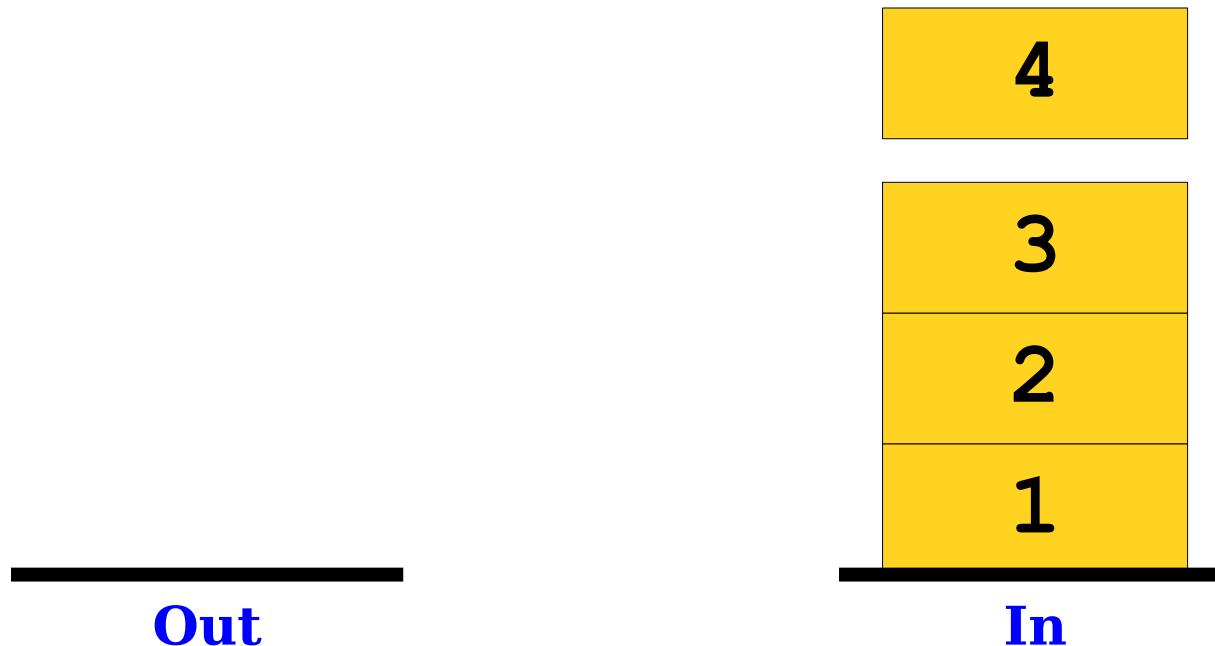
Φ = height of **In** stack



$$\begin{aligned} \text{amortized-cost} &= \text{real-cost} + k \cdot \Delta\Phi \\ &= O(1) + k \cdot 1 \\ &= \mathbf{O(1)} \end{aligned}$$

The Two-Stack Queue

Φ = height of **In** stack



The Two-Stack Queue

Φ = height of **In** stack



$$\begin{aligned} \text{amortized-cost} &= \text{real-cost} + k \cdot \Delta\Phi \\ &= O(h) + k \cdot -h \quad // h = \text{height of } \mathbf{In} \text{ stack} \\ &= \mathbf{O(1)} \quad // \text{Choose } k \text{ strategically} \end{aligned}$$

The Two-Stack Queue

Φ = height of **In** stack



$$\begin{aligned} \text{amortized-cost} &= \text{real-cost} + k \cdot \Delta\Phi \\ &= O(1) + k \cdot 0 \\ &= \mathbf{O(1)} \end{aligned}$$

Why This Works

$$\sum \text{amortized-cost} = \sum (\text{real-cost} + k \cdot \Delta \Phi)$$

Why This Works

$$\begin{aligned}\sum \text{amortized-cost} &= \sum (\text{real-cost} + k \cdot \Delta\Phi) \\ &= \sum \text{real-cost} + k \cdot \sum \Delta\Phi\end{aligned}$$

Why This Works

$$\begin{aligned}\sum \text{amortized-cost} &= \sum (\text{real-cost} + k \cdot \Delta\Phi) \\ &= \sum \text{real-cost} + k \cdot \sum \Delta\Phi \\ &= \sum \text{real-cost} + k \cdot (\Phi_{end} - \Phi_{start})\end{aligned}$$

Why This Works

$$\begin{aligned}\sum \text{amortized-cost} &= \sum (\text{real-cost} + k \cdot \Delta\Phi) \\ &= \sum \text{real-cost} + k \cdot \sum \Delta\Phi \\ &= \sum \text{real-cost} + k \cdot (\Phi_{end} - \Phi_{start}) \\ &\geq \sum \text{real-cost}\end{aligned}$$

Let's make two assumptions:

$$\begin{aligned}\Phi &\geq 0. \\ \Phi_{start} &= 0.\end{aligned}$$

The Story So Far

- We will assign amortized costs to each operation such that

$$\sum \text{amortized-cost} \geq \sum \text{real-cost}$$

- To do so, define a **potential function** Φ such that
 - Φ measures how “messy” the data structure is,
 - $\Phi_{start} = 0$, and
 - $\Phi \geq 0$.
- Then, define amortized costs of operations as

$$\text{amortized-cost} = \text{real-cost} + k \cdot \Delta\Phi$$

for a choice of k under our control.

Theorem: The amortized cost of any enqueue or dequeue operation on a two-stack queue is $O(1)$.

Proof: Let Φ be the height of the *In* stack in the two-stack queue. Each enqueue operation does a single push and increases the height of the *In* stack by one. Therefore, its amortized cost is

$$O(1) + k \cdot \Delta\Phi = O(1) + k \cdot 1 = O(1).$$

Now, consider a dequeue operation. If the *Out* stack is nonempty, then the dequeue does $O(1)$ work and does not change Φ . Its cost is therefore

$$O(1) + k \cdot \Delta\Phi = O(1) + k \cdot 0 = O(1).$$

Otherwise, the *Out* stack is empty. Suppose the *In* stack has height h . The dequeue does $O(h)$ work to pop the elements from the *In* stack and push them onto the *Out* stack, followed by one additional pop for the dequeue. This is $O(h)$ total work.

At the beginning of this operation, we have $\Phi = h$. At the end of this operation, we have $\Phi = 0$. Therefore, $\Delta\Phi = -h$, so the amortized cost of the operation is

$$O(h) + k \cdot -h = O(1),$$

assuming we pick k to cancel out the constant factor hidden in the $O(h)$ term. ■

Why This Works

$$\begin{aligned}\sum \text{amortized-cost} &= \sum (\text{real-cost} + k \cdot \Delta\Phi) \\ &= \sum \text{real-cost} + k \cdot \sum \Delta\Phi \\ &= \sum \text{real-cost} + k \cdot (\Phi_{end} - \Phi_{start}) \\ &\geq \sum \text{real-cost}\end{aligned}$$

Let's make two assumptions:

$$\begin{aligned}\Phi &\geq 0. \\ \Phi_{start} &= 0.\end{aligned}$$

Potential Method



Idea : Take a closer look at the *credit* --

$$\hat{c}_i - c_i = \text{Credit}_i = \Phi(D_i) - \Phi(D_{i-1})$$

Potential function

$$\begin{aligned}\sum_{i=1}^n \hat{c}_i &= \sum_{i=1}^n (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \left(\sum_{i=1}^n c_i \right) + \underbrace{\Phi(D_n) - \Phi(D_0)}_{\geq 0}\end{aligned}$$

Should be bounded.

In general, a good potential function should always assume its minimum at the start of the sequence.

Tips: The most complicated operation usually leads to a significant decrease in the potential function.

【Example】 Stack with **MultiPop(int k, Stack S)**

D_i = the stack that results after the i -th operation

$\Phi(D_i)$ = the number of objects in the stack D_i

$$\Phi(D_i) \geq 0 = \Phi(D_0)$$

Push: $\Phi(D_i) - \Phi(D_{i-1}) = (\text{sizeof}(S) + 1) - \text{sizeof}(S) = 1$

$$\rightarrow \hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$$

Pop: $\Phi(D_i) - \Phi(D_{i-1}) = (\text{sizeof}(S) - 1) - \text{sizeof}(S) = -1$

$$\rightarrow \hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 - 1 = 0$$

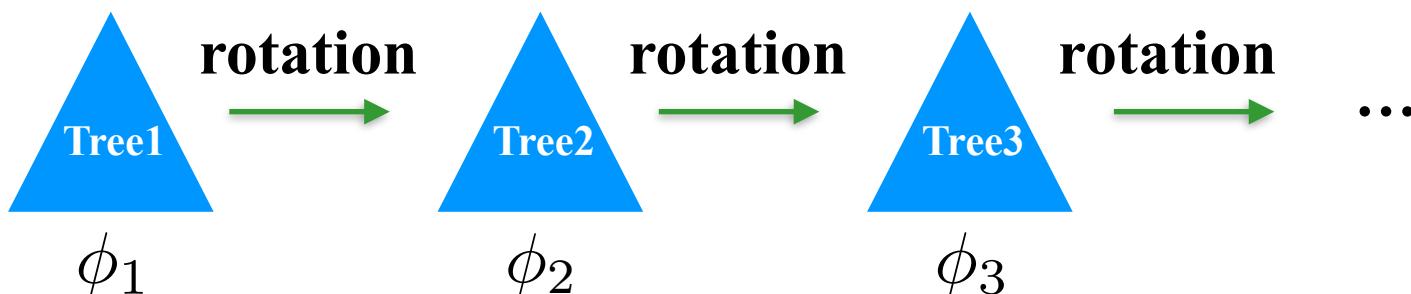
MultiPop: $\Phi(D_i) - \Phi(D_{i-1}) = (\text{sizeof}(S) - k') - \text{sizeof}(S) = -k'$

$$\rightarrow \hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k' - k' = 0$$

$$\sum_{i=1}^n \hat{c}_i = \sum_{i=1}^n O(1) = O(n) \geq \sum_{i=1}^n c_i \rightarrow T_{\text{amortized}} = O(n)/n = O(1)$$

Analysis of Splay Trees

- What do we want to bound?
 - The amortized cost of a sequence of operations, e.g. search, delete, insert, split...
 - Each operation involves splaying: a subsequence of rotations.
- The potential function is built on one state of tree. Let's consider the amortized cost of one splay process (a sequence of rotations) first.



【Example】 Splay Trees: $T_{\text{amortized}} = O(\log N)$

D_i = the root of the resulting tree

$\Phi(D_i)$ = must increase by at most $O(\log N)$ over n steps, AND will also cancel out the number of rotations (zig:1; zig-zag:2; zig-zig:2).

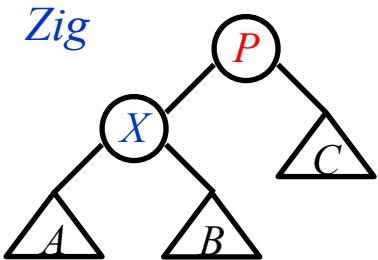
$$\Phi(T) = \sum_{i \in T} \log S(i) \quad \text{where } S(i) \text{ is the number of descendants of } i \text{ (} i \text{ included).}$$

Rank of the subtree
 \approx Height of the tree

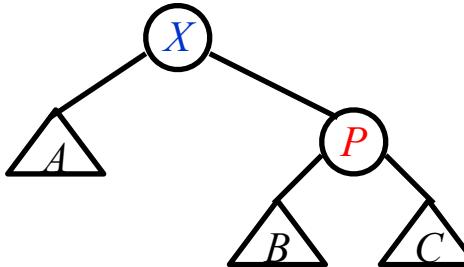
Why not simply use the heights
of the trees?



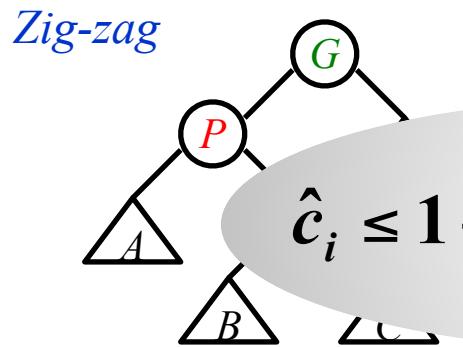
$$\Phi(T) = \sum_{i \in T} Rank(i)$$



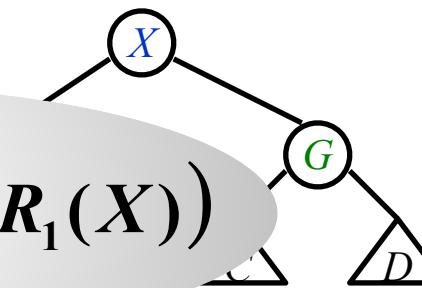
Single rotation



$$\begin{aligned}\hat{c}_i &= 1 + R_2(X) - R_1(X) \\ &\quad + \underline{R_2(P) - R_1(P)} \\ &\leq 1 + R_2(X) - R_1(X)\end{aligned}$$

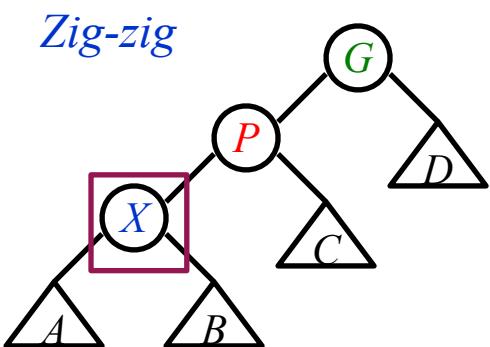


Double rotation

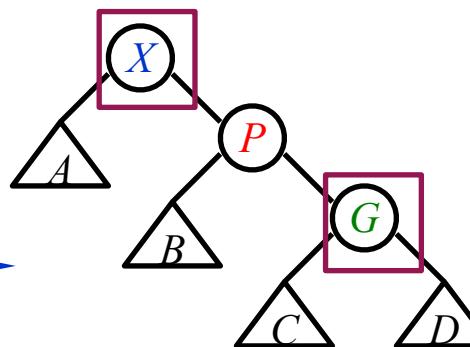


$$\begin{aligned}\hat{c}_i &= 2 + R_2(X) - \underline{R_1(X)} \\ &\quad + R_2(P) - \underline{R_1(P)} \\ &\quad + R_2(G) - \underline{R_1(G)} \\ &\leq 2(R_2(X) - \underline{R_1(X)})\end{aligned}$$

Lemma 11.4 on [Weiss] p.448



Single rotation



$$\begin{aligned}\hat{c}_i &= 2 + R_2(X) - R_1(X) \\ &\quad + R_2(P) - R_1(P) \\ &\quad + R_2(G) - R_1(G) \\ &\leq 3(R_2(X) - R_1(X))\end{aligned}$$

[Lemma] The total cost of $\sum \hat{c}_i$ to splay a tree by a series of rotations with root T at node X is at most $3(R(T) - R(X)) + 1$.

Amortized Cost of Splay Trees

【Lemma】 The total cost of $\sum \hat{c}_i$ to splay a tree by a series of rotations with root T at node X is at most $3(\underline{R(T)} - R(X)) + 1$.

$$\begin{aligned} \sum_{i=1}^n \hat{c}_i &= \sum_{i=1}^n (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \left(\sum_{i=1}^n c_i \right) + \frac{\Phi(D_n) - \Phi(D_0)}{\geq 0} \end{aligned}$$

Should assume
to start from
an empty tree

We should also consider the influences of other steps other than rotations on the potential functions.

Fortunately, their influences are minor.

Theorem:

The amortized cost of a series of operations started from an empty splay tree is of order $O(\log N)$, where N is the number of all nodes involved in the operations.

Read the original splay tree paper for details.

Balanced Binary Search Trees (I)

- Binary search trees
- AVL trees
- Splay trees
- Amortized analysis
- Take-home messages

Take-Home Messages

- Balanced binary search trees:
 - Reduce depth to reduce cost of operations.
- AVL trees:
 - Satisfying height-balanced condition. Conduct rotations to achieve self-balancing once the condition is violated.
- Splay trees:
 - Achieving self-balancing by conducting splaying steps for any operations.
- Amortized analysis:
 - Amortizing the total cost which is limited by the structure.

$$\text{amortized-cost} = \text{real-cost} + k \cdot \Delta\Phi.$$

Thanks for your attention!
Discussions?

Reference

- Data Structure and Algorithm Analysis in C (2nd Edition): [Chap. 4.4-4.5, 11.5.](#)
- Introduction to Algorithms (4th Edition): [Chap.16.](#)
- Daniel Dominic Sleator, Robert Endre Tarjan:
Self-Adjusting Binary Search Trees. Journal of ACM 32(3): 652-686 (1985)
- <https://web.stanford.edu/class/archive/cs/cs166/cs166.1206/lectures/07/Small07.pdf>
- <https://ocw.mit.edu/courses/6-046j-design-and-analysis-of-algorithms-spring-2015/>
(Lecture 5: Amortized analysis)
- <https://ocw.mit.edu/courses/6-006-introduction-to-algorithms-spring-2020/>
(Lecture 7: AVL tree)